Improved Vertex Skinning Algorithm based on Dual Quaternions

A thesis
submitted in partial fulfilment
of the requirements for
the Degree of Master of Science
in Computer Science and Software Engineering
by

Hao Yin

Dr R. Mukundan ..... Supervisor
Associate Professor, Department of Computer Science and
Software Engineering
University of Canterbury

2019

To grandpa.

Abstract

In character animation, traditional skinning algorithms such as linear blend skinning (LBS) and dual quaternion skinning (DQS) are widely used due to their simplicity and efficiency. However, they both suffer from some well-known issues. Discovering solutions to these artifacts involves ongoing research with the same new developments and applications. In this paper, we propose an improved algorithm to reduce the artifact of standard DQS by combining it with LBS. Our algorithm calculates a linear combination of the standard DQS and LBS using a parameter, called the "deform factor". The "deform factor" describes the contributions of DQS and LBS in the skinning process. Our improved skinning algorithm is based on dual quaternions, and it effectively reduces the artifacts of standard DQS. Furthermore, the proposed method presents a visual trade-off between DQS and LBS in real-time. This thesis also contributes an extensive experimental analysis showing the robustness of the proposed method and a comparison of its performance relative to the traditional approaches.

Acknowledgments

Firstly, it is my pleasure to thank my supervisor Dr. R. Mukundan for his guidance and insightful comments during my thesis. His support and knowledge have been of great help to me and my research. I want to pay tribute to colleagues from the Department of Computer Science and Software Engineering for helping me throughout my studies. I also would like to thank everyone in the Computer Graphics and Medical Image Analysis research group, who have no doubt made the year an enjoyable one. I appreciate the digital team of Fulton Hogan IT, especially Mrs. A. Lawrey, for all the support. Finally, I must pay the biggest gratitude to my family, my friends, especially my parents and my wife, Sabrina, for constant encouragement, support, and have been there for me when the going was tough.

Table of Contents

List of Figures

List of Tables

Chapter I

Introduction

Character animation is the art of bringing 2D and 3D characters to life, and it adds kinematic motions, the illusion of thoughts and even personality to a virtual character. In production computer animation pipelines, character animation is a vital component and a specialised area of the animation process. Skeleton-based animation (or skeletal animation) is the most common approach in computer character animation. The skeleton is a set of interconnected bones used to move the surface representation (called skin or mesh) of a computer character. While this approach is suited mostly to humanoid models, it is also suitable for animating other objects such as cars, trees, and animals the equivalent way as long as the skin and the skeleton are defined.

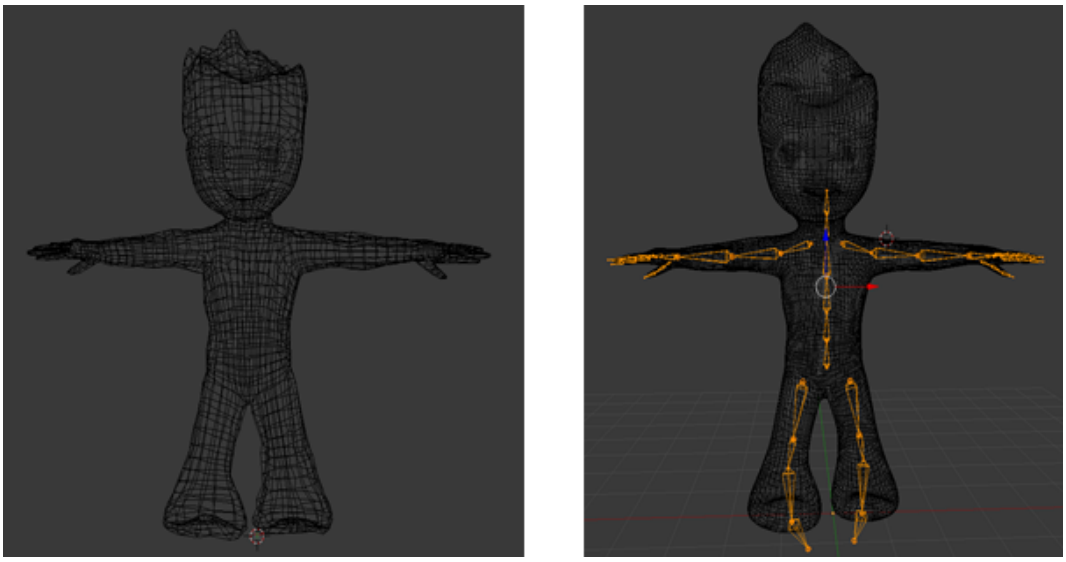


Figure 1.1. A complete mesh (left) of a humanoid character "Groo" in rest pose. Rigging (right), create a hierarchical set of bones.

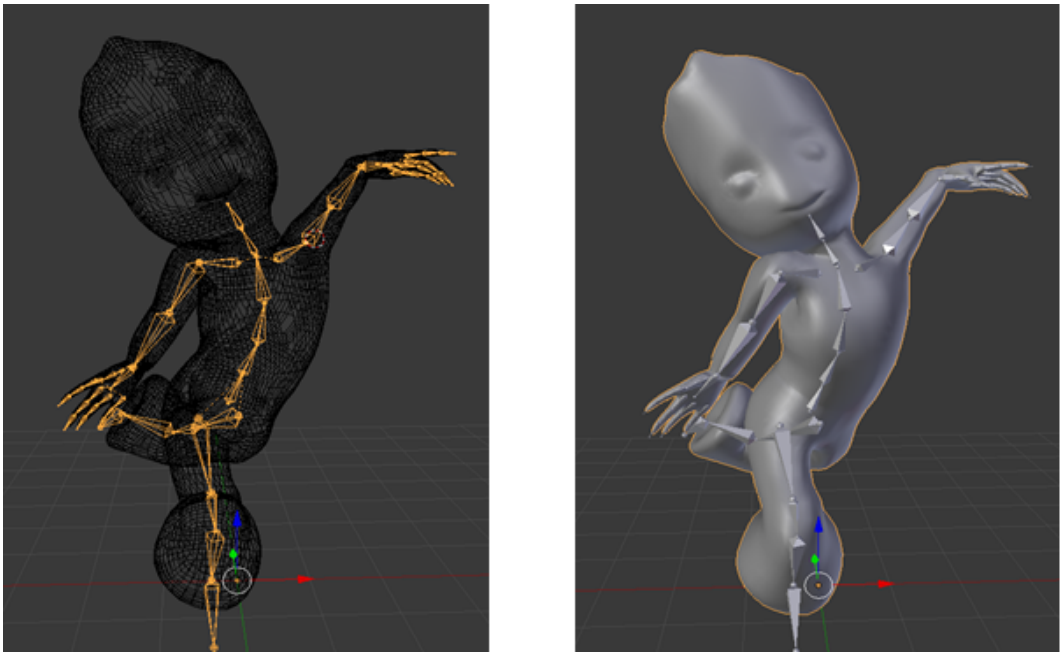


Figure 1.2. Skinning and animating "Groo" in Blender. Skinning (left), a process of creating the vertex-bone bindings and move the mesh with associated bones. Keyframes animation (right), contains a sequence of single frames of the animating character.

Usually, skeletal animation consists of three steps. First, rigging (see figure 1.1) is the process of setting up the hierarchy of bones with controls according to the rules of kinematics. Second, skinning (see figure 1.2 left) is a process of associating each bone with some portion of the character's skin. Lastly, keyframes animation (see figure 1.2 right) defines the start point and end point of a sequence of smooth transitions which represent a complete animation. Both rigging and keyframes animation can be conveniently done in 3D graphics tools such as Blender, Autodesk Maya, and 3ds Max by 3D artists with expertise. The process of skinning is different from the other two, which generally requires effort from computer graphics developers. Graphics developers implement various skinning techniques that can sufficiently deform the model mesh and satisfy different animation requirements. In most computer animation pipelines, skinning techniques play an essential role. Various skinning algorithms are implemented in the graphics tools mentioned above or rendering engines and are frequently used by 3D artists to create animations.

To achieve high-quality and believable effects, graphics researchers have invented skinning techniques and enhanced with the recent development of both computer graphics hardware and software. Traditional skinning techniques such as linear blend skinning (LBS) and dual quaternion skinning (DQS) are the de-facto standard and most widely used, but advanced skinning techniques such as example-based and physics-based skinning are receiving popularity in the industry increasingly. Commonly, traditional skinning techniques are compulsory for animation pipelines and 3D graphics tools to produce basic character animation in real-time. In contrast, advanced methods are often explicitly implemented in game engines or high-performance renderers to satisfy higher-level requirements. Compared with traditional skinning methods, advanced techniques require more significant computa-

tional resources than conventional techniques to generate more realistic and complex deformations.

Interestingly, even though advanced skinning techniques produce a more highly detailed deformation that traditional ones are not capable of, traditional skinning techniques are still the most popular method at present as they are straightforward and efficient. Traditional skinning techniques are considerably more suitable for modern GPU rendering, which is often ideal for real-time interactive applications. Although recent blooming in machine learning technology boosts the performance for advanced skinning techniques, traditional methods are still advantageous as they are straightforward to implement and easy to understand.

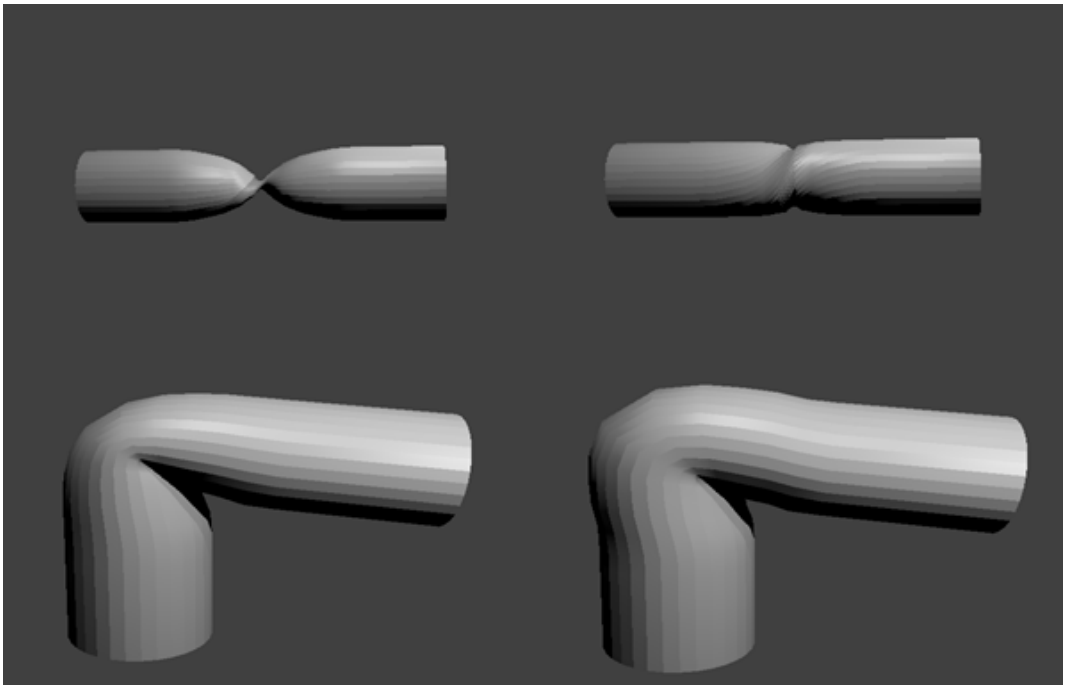


Figure 1.3. Animating a cylinder model in Blender. (Left column) Twisting and bending the cylinder with Blender's default skinning algorithm: Linear blend skinning. (Right column) Same animation but with dual quaternion skinning in Blender.

However, their simplicity and efficiency mean some level of loss of ac-

curacy with resulting deformation. Traditional methods are constrained by the limitation of being unable to produce secondary skin effects, such as muscle bulging and wrinkles, and they suffer from many artifacts. LBS has the well-known "collapsing joint" and "candy-wrapper" artifact: in the left column of figure 1.3, twisting the cylinder by nearly 180 degrees reveals the "candy-wrapper" artifact, and bending the cylinder by 90 degrees shows the "collapsing joint" effect. DQS overcomes the LBS artifacts but also suffers from its own "bulging joint" artifact. In the right column of figure 1.3, twisting (top) with DQS solves the "candy-wrapper" artifact, but bending (bottom) with DQS bulges the mesh at the joint.

This thesis conducts a study on the traditional skinning techniques in particular. Specifically, we focus on DQS and propose an improved skinning method based on dual quaternions. Since dual quaternions can only represent rigid transformations, the homogeneous transformations applied in this study are assumed to be rigid only, and non-rigid transformations are out of the scope of this thesis. In the following part of this chapter, the aim of the dissertation is explained in section 1.1. A general description of the relevant materials and methodology of this thesis is given in section 1.2. Section 1.3 defines the thesis motivation. Section 1.4 lists some prior knowledge that would support reading this thesis. Finally, section 1.5 summarises this chapter followed by an overview structure of the whole dissertation.

1.1 Aims of the thesis

This thesis aims to develop an improved vertex skinning algorithm based on dual quaternions. The improved skinning algorithm is expected to reduce the "bulging joint" artifact of the standard DQS while retaining the same level of simplicity and efficiency. A GPU implementation of the proposed method without multi-phase processing or altering the skin geometry aims to plug

into an existing standard forward kinematics pipeline. Also, it is hypothesised that our approach will provide sufficient flexibility and maintain a high level of system robustness.

1.2 Materials and methods

1.2.1 Materials relevant to the thesis

This thesis uses several 3D character models. The models are required to be rigged, with pre-defined skinning weights attached to the skeleton. Character models are obtained from open source 3D model providers Mixamo and Sketchfab. The models obtained have different levels of complexity, in terms of the mesh surface geometry and skeleton structure. Each model also contains a different set of keyframes animation, with different lengths of animation sequences.

Furthermore, some specific poses are obtained and applied to the models. The poses contain extreme joint rotations that are suitable for the evaluation of the proposed vertex skinning technique. Several models have textures attached, to examine how the proposed skinning technique acts on textures.

To visualise the skinning result, a complete character animation system is implemented with the modern OpenGL API (OpenGL 4.5) in this dissertation. In particular, the proposed run-time vertex skinning algorithm is performed in the OpenGL shader pipeline, which is a common approach in computer animation nowadays.

Materials relevant to the proposed vertex skinning algorithm include quaternions, dual quaternions, and skeleton animation concepts, which are discussed in chapter 2.

1.2.2 Research methodology

We implemented an interactive character animation system that takes character models as input and deforms the model mesh with an improved vertex skinning algorithm. The implementation of the vertex skinning algorithm leverages the power of GPU hardware to ensure the real-time performance of the animation system.

The proposed vertex skinning algorithm combines standard LBS and DQS in the linear space in real-time. A parameter called the "deform factor" is introduced to alter the influence of each standard technique during the deformation of the model mesh, thus reducing the DQS artifacts by bringing back portions of features from LBS. The following scheme (see figure 1.4) illustrates the pipeline of the implementation:

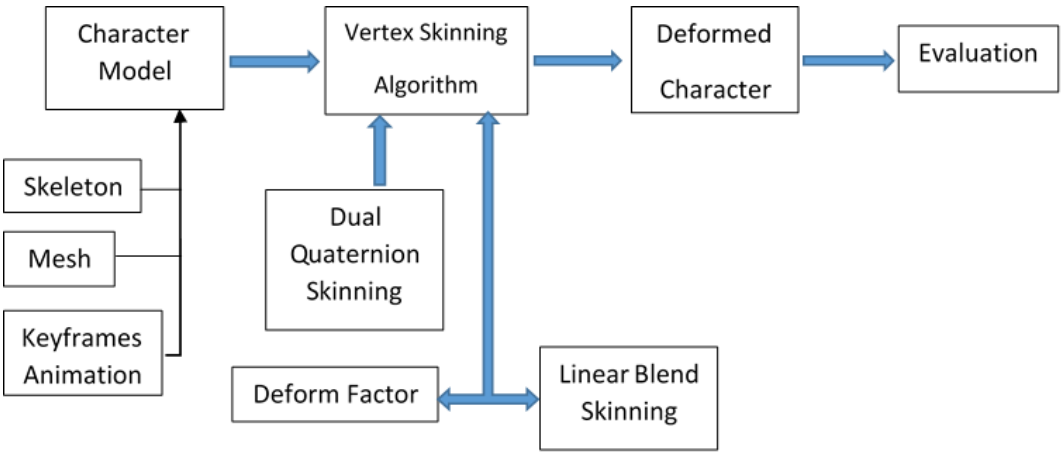


Figure 1.4. The pipeline of implementation

The proposed vertex skinning algorithm linearly combines DQS and LBS giving a complete character model with a single mesh model, a skeleton and an animation sequence. The amount of LBS influence is determined by the "deform factor" via a UI component. As a result, the model mesh is deformed by a mixture of DQS and LBS influences, thus reducing the

"bulging joint" artifact of DQS.

Similarly, Maya's Blend Smooth Skinning [5] method provides users with the ability to combine the two standard skinning algorithms linearly. However, artists are required to paint a blend weight map in advance. This process is iterative, combining expertise with trial and error, and sometimes it can be very tedious to achieve the expected quality. Compared to Maya's Blend Smooth Skinning, the proposed method combines DQS and LBS without any dependencies on an extra weight map. Besides, it provides users with the ability to adjust the influence of standard techniques at run-time easily.

The evaluation of the proposed method contains both qualitative and quantitative analysis. In particular, the proposed vertex skinning technique is evaluated subjectively from the quality of the resulting deformation against the standard DQS and LBS among various 3D characters and animations. The evaluation includes a visual examination of the deformed character at challenging areas with specific poses and comparisons with DQS and LBS artifacts. The efficiency and robustness of the proposed method are evaluated quantitatively from the performance and memory requirements against the standard DQS and LBS. This analysis includes frame rate counts and buffer memory allocation comparison.

1.3 Motivations

The first publication of DQS occurs in 2007, and the method gradually becomes frequently used in the industry. Compared to LBS, DQS requires more complicated calculations. Nevertheless, there are not many related learning materials about DQS available online, which makes it challenging for junior graphics developers to understand this technique. The challenge was the motivation to investigate dual quaternions and build an implementation on it.

To the best of our knowledge, currently only Maya supports a linear blend of the two standard skinning algorithms (LBS and DQS). Maya's configuration lacks systematic studies and analysis. Moreover, the concept of a linear combination of LBS and DQS had almost no prior research. For this reason, this thesis was motivated to implementing this concept and conduct a systematic study to fill the gap in the literature.

1.4 Assumptions of prior knowledge

This thesis assumes that the reader has a certain amount of prior knowledge in the areas of modern computer graphics and mathematics. On the computer graphics side, the reader is assumed to have some knowledge of the modern graphics pipeline with OpenGL graphics API and GPU programming such as the use of shaders. In particular, some prior experience of vertex and fragment shader programming would be of help. Although some previous exposure to skeleton animation with ASSIMP and traditional skinning algorithms such as LBS and DQS would be an advantage, this document assumes no prior knowledge of these and aims to explain them thoroughly from the ground up. On the mathematics side, a familiarity with linear algebra, complex numbers, dual numbers, and transformation with matrices is assumed.

1.5 Structure of the thesis

This thesis is made up of five chapters. Chapter 1 introduces the context of our research and explains the motivation and research goals. In chapter 2, a brief background study of the key concepts that are applied in this thesis is presented. Chapter 3 reviews state of the art in skinning. Chapter 4 describes the design and implementation of the proposed method and the

character animation system. Chapter 5 presents the vertex skinning results and compares the result of the proposed method with other approaches. The thesis finishes with chapter 6 to summarise and discuss future directions.

Chapter II

Key Concepts

This chapter outlines the background of this dissertation and defines several related vital concepts. Section 2.1 and 2.2 explains quaternion and dual quaternion theory and their application in computer graphics. Section 2.3 focus on describing the knowledge of the skeleton animation. Including mesh models, bones, vertex skinning concepts, and keyframes animation.

2.1 Quaternions

The discovery of quaternions arose historically from Hamiltons attempts in 1843. Quaternions satisfy the need for comprehensive methods of representing orientation and provide significant advantages over the traditional way of using Euler angles to define orientations. Moreover, quaternions are extremely useful for interpolating between orientations in the 3D space. The most widespread use of quaternions to date is in computer animation. They are used to represent transformations of the orientation of graphical objects. Quaternions have also been present in quantum mechanics and are closely related to "spinors".

The construction of quaternions is based on the theory of complex numbers. A quaternion is defined as:

$$q = w + (xi + yj + zk) \tag{2.1}$$

where w, x, y, z are real numbers and i, j, k are the imaginary components. The imaginary components satisfy $i^2 = j^2 = k^2 = ijk = -1$, and they are not communicative. The component w is also known as the scalar part of a quaternion, and x, y, z are also known as the vector part. It is more common in computer graphics to represent a quaternion as these two components:

$$q = (w, v) \tag{2.2}$$

Derived from Kenwright's detailed description on quaternions [6], the elementary quaternion arithmetic operations are used in this thesis and can, therefore, express with this representation as below:

- Scalar multiplication:

$$sq = (sw, sv), \text{where s is a scalar parameter} \tag{2.3}$$

- Addition:

$$q_1 + q_2 = (w_1 + w_2, v_1 + v_2) \tag{2.4}$$

- Multiplication, is defined by the Hamilton product:

$$q_1 q_2 = (w_1 w_2 - v_1 \cdot v_2, w_1 v_2 + w_2 v_1 + (v_1 \times v_2)) \tag{2.5}$$

- Conjugate:

$$q^* = (w, -v) \tag{2.6}$$

- Magnitude:

$$\|q\| = \sqrt{qq^*} \tag{2.7}$$

In a 3D rotation, transformations usually correspond to unit quater-

nions. The unit quaternion is a quaternion whose length equals to 1, which can be calculated by dividing each of its components by the magnitude. The scalar component of the unit quaternion represents the amount of rotation which will occur, and the vector part represents the axis about the rotation.

Unit quaternions can be used to perform the rotational transformation to a point in 3D space. Given a unit quaternion Q and a point $v = (x, y, z)$, the rotation of the point is calculated as:

$$v' = q(1 + xi + yj + zk)q^* \tag{2.8}$$

where v' is the transformed point. Unit quaternion transformations are written in the conventional homogeneous transformation matrix form using the four real components as shown below:

$$\begin{bmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy & 0 \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz+2wx & 0 \\ 2xz-2wy & 2yz-2wx & 1-2x^2-2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

However, quaternions are not an equivalent representation of 3D rotations compared to matrices. It is because two quaternions, q and $-q$, represent the same rotations, which is called the "double cover" property of quaternions by Hanson [7].

Quaternion interpolation is an important property, and two quaternions can be interpolated continuously to form a curved-manifold for blending rotations. A basic technique of quaternion interpolation is called quaternion linear interpolation (LERP or QLERP), which linearly interpolates the unit quaternions. Given two unit quaternions, q_1 and q_2, a linear interpolation

results in the quaternion:

$$q = (1 - t)q_1 + tq_2 \tag{2.10}$$

where t is the interpolation parameter that satisfies $0 \leq t \leq 1$. However, the resulting distribution is unevenly spread on the arc of the unit sphere and causes non-uniformity in the angular velocity from one quaternion to the other. Shoemake [8] presents another algorithm for interpolating two quaternions called the spherical linear interpolation (SLERP) technique. SLERP computes intermediate quaternions in a unit sphere between two unit quaternions and generates a uniformly interpolated rotation of the object from one orientation to another with constant angular velocity. The SLERP formula is given as:

$$q = \frac{q_1 \sin\left((1 - t)\theta\right) + q_2 \sin\left(t\theta\right)}{\sin\theta} \tag{2.11}$$

where θ is the angle between q_1 and q_2. Interpolation between two quaternions is likely to produce a path that corresponds to a greater than 180 degrees rotation due to the "double cover" property.

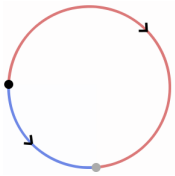


Figure 2.1. The antipodality of quaternions

As illustrated in figure 2.1, the blue trajectory represents the transformation of the point on the sphere produced by a counter-clockwise rotation (shortest path), while the red trajectory represents the same transformation produced by a clockwise rotation (longer path). To ensure SLERP provides the shortest path between two endpoints, and maps to a rotation through

the angle between the quaternions, the correct sign of the quaternion is important. Specifically, if the angle between q_1 and q_2 is less than 90 degrees, the interpolation is between two quaternions. Otherwise, the interpolation is between q_1 and $-q_2$. The process of correcting the signs of quaternions is also called antipodality handling in vertex skinning [9], and the operation of flipping the quaternion sign is denoted as $\oplus$ [10].

2.2 Dual quaternions

Dual quaternions extend quaternions with the dual numbers theory introduced by Clifford [11]. The dual numbers consist of two components:

$$z = r + \varepsilon d \tag{2.12}$$

where element ε is the dual operator with $\varepsilon^2 = 0$ and $\varepsilon \neq 0$, r is the real part (or non-dual part) and d is the dual part.

A dual quaternion was first presented as Clifford algebra comprised of two quaternions [11], which not only have similar elementary arithmetic as quaternions and represent rotations but also include information about translation, which was missing in quaternions. Each dual quaternion consists of two quaternions, and they are distinguished as the real part and dual part according to the dual numbers theory. If the real part is denoted as q_r and the dual part is denoted as q_d, then a dual quaternion $\hat{q}$ is defined as:

$$\hat{q} = q_r + q_d\varepsilon = (r_w, r_x, r_y, r_z) + (d_w, d_x, d_y, d_z)\varepsilon \tag{2.13}$$

where ε is the dual unit, (r_w, r_x, r_y, r_z) and (d_w, d_x, d_y, d_z) are the quaternion representation of the real part and the dual part. The elementary arith-

metic operations of dual quaternions that are used in this thesis are given as [6]:

- Scalar multiplication:

$$s\hat{q} = sq_r + sq_d\varepsilon \tag{2.14}$$

- Addition:

$$\hat{q}_1 + \hat{q}_2 = q_{r1} + q_{r2} + (q_{d1} + q_{d2})\varepsilon \tag{2.15}$$

- Multiplication:

$$\hat{q}_1\hat{q}_2 = q_{r1}q_{r2} + (q_{r1}q_{d2} + q_{d1}q_{r2})\varepsilon \tag{2.16}$$

- Conjugate:

$$\hat{q}^* = q_r^* + q_d^*\varepsilon \tag{2.17}$$

- Magnitude:

$$\|\hat{q}\| = \sqrt{\hat{q}\hat{q}^*} \tag{2.18}$$

Similar to unit quaternions, unit dual quaternions can also represent 3D transformations. Transformations using unit dual quaternions are given as:

$$p' = \hat{q}p\hat{q}^* \tag{2.19}$$

where p and p' represent the transforming point in 3D space embedded in a unit quaternion, and $\hat{q}$ and $\hat{q}^*$ represent a dual quaternion and its conjugate.

The rotational information of a unit dual quaternion corresponds to the real part quaternion q_r and the dual part quaternion q_d represents the translational information. Given a translation vector $\vec{t} = (t_0, t_1, t_2)$ in three-dimensional space, The translational information of the unit dual quaternion

in the form of a unit quaternion is given as [12]:

$$q_d = (0, \frac{t_0}{2}, \frac{t_1}{2}, \frac{t_2}{2}) \tag{2.20}$$

Thus a unit dual quaternion $\hat{q}$ with no rotation is represented as follows:

$$\hat{q} = (1, 0, 0, 0, 0, \frac{t_0}{2}, \frac{t_1}{2}, \frac{t_2}{2}) \tag{2.21}$$

Recalling the matrix form of unit quaternions, the matrix form of a unit dual quaternion is essentially a compound of the rotation matrix that corresponds to the real part unit quaternion and the translation vector. The homogeneous transformation matrix form of dual quaternions is given as [13]:

$$\begin{bmatrix} r_w^2 + r_x^2 - r_y^2 - r_z^2 & 2r_x r_y - 2r_w r_z & 2r_x r_z + 2r_w r_y & t_0 \\ 2r_x r_y + 2r_w r_z & r_w^2 - r_x^2 + r_y^2 - r_z^2 & 2r_y r_z - 2r_w r_x & t_1 \\ 2r_x r_z - 2r_w r_y & 2r_y r_z + 2r_w r_x & r_w^2 - r_x^2 - r_y^2 + r_z^2 & t_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.22}$$

Dual quaternions can be used to generalise the quaternion blending algorithms and deal with all rigid transformations. Kavan et al. [14] present three popular approaches — the screw linear interpolation (ScLERP), the dual quaternion linear blending (DLB), and the dual quaternion iterative blending (DIB). ScLERP simulates the screw motion for interpolation, which involves rotation and a translation of the same screw axis. Both the translation and rotation are linearly interpolated around the screw axis. Essentially, ScLERP is an extension of SLERP, hence the produced interpolation ensures a constant angular velocity with the shortest path. However, the same as SLERP, ScLERP lacks the support of more than two rigid transformations.

The DLB algorithm is more commonly used in dual quaternion based

character animation systems as it is extremely simple. The DLB algorithm computes a blended unit dual quaternion as follows:

$$DLB(w; \hat{q_1}, ..., \hat{q_n}) = \frac{w_1\hat{q_1} + ... + w_n\hat{q_n}}{\|w_1\hat{q_1} + ... + w_n\hat{q_n}\|} \tag{2.23}$$

where the blending parameters $w_{1...n}$ are a set of convex weights. The algorithm calculates a weighted sum of unit dual quaternions followed by a normalisation, which extends from QLERP. Hence, the resulting interpolation is not perfect manifold-intrinsic averaging. However, the DLB algorithm is capable of more than two rigid transformations, and the imprecision of DLB is most likely not be visible at all compared to the more accurate ScLERP [9]. Due to dual quaternions inheriting the "double cover" property of quaternions, the appropriate signs of dual quaternion are also required to ensure the shortest path of interpolation [9].

The DIB algorithm blends rigid transformations in subsequent iterations. The algorithm not only follows the properties of DLB, but it is also constant speed and shortest path. Kavan's work [14] proves the result of DIB is equivalent to the outcome of ScLERP with only a single iteration. Therefore it represents the exact rigid transformation blending. The drawback of the DIB algorithm is the computation can be more complex and slower than the DLB algorithm.

Compared to rigid transformation using matrices, dual quaternions provide a more efficient alternative in terms of computational cost by requiring four fewer floats (3×4 matrix requires 12 floats compared to a dual quaternion that needs eight floats). In computer animation, the advantages of dual quaternion interpolation over matrices have been demonstrated with the well-known dual quaternion skinning (DQS) method [9]. Furthermore, dual quaternions have been widely used in the autonomous robotics field

to solve inverse kinematic problems from dual quaternion feedback [15] and represent body models [16].

2.3 Skeleton animation

This section explains the fundamental components of skeleton animation, such as mesh models, bones, vertex skinning, and keyframes animations. The concepts and materials in this section are used throughout the thesis and are the key building blocks of the implementation part.

2.3.1 Mesh models

Mesh models are those, in the specific case of 3D models, built from a single or multiple polygon meshes. Mesh models contain a collection of vertices, which are defined using a common coordinate system. Vertices of a mesh form some faces. Two primitive shapes of the face are triangular and quadrilateral. Information from vertices and faces as a whole define the overall surface geometry of the mesh model.

A mesh is a 3D shape of a model, and the texture is a sheet lying on the surface of the model. A texture is a rectangular image that covers the mesh at specific vertices. Textures provide the models with colours, normals, and reflectivity. Textures are also called materials of mesh models.

Mesh models are used to represent real-world objects in the 3D world, including but not limited to humanoid objects. Computer graphics artists normally do the creation of a mesh model. Mesh models are often available on 3D model platforms such as TurboSquid and Sketchfab. Model files can have various formats, and common ones include OBJ, X, PLY, DAE/Collada, 3DS, and FBX.

Skeleton animation systems treat mesh models as input data. The Open Asset Import Library (ASSIMP) is useful for model importing regard-

less of file formats. ASSIMP loads several input model formats into one straightforward data structure for further processing. There are also tools for processing and manipulating mesh models, such as MeshLab, OpenMesh, and OpenVDB.

2.3.2 Bones

Bones are the underlying skeleton of the mesh model. A bone is an abstract data structure of the model, not a graphics primitive. Each bone groups the entire mesh into different body parts, such as head, chest, and the left hand of a human model. A bone contains information about its position and orientation relative to its parent. The complete set of the bones, along with the connectivity information, make up a hierarchical skeleton. Such an initial configuration defines the rest pose (or bind pose) of the skeleton. The bone offset matrix defines the transformation from the bone's local coordinate space to the skeleton's coordinate space.

In skeleton animation, each bone deforms a space around it. The part of the mesh belonging to a bone is called the skin of the bone. The position and orientation of a joint determine the bone transformations that furthermore moves the associated mesh. Usually, vertices on the related mesh are first transformed back to the bone's local coordinate using the bone's offset matrix, and then they apply the joint angle transformation to return to the skeleton space. This process is fundamental to skeleton animation.

Bone transformations are regularly affine, and they can be rigid and non-rigid. Rigid transformations include rotational and translational information, while non-rigid transformations refer to scale information. Although non-rigid transformation can provide additional deformation, such as muscle bulges, it is more common to apply rigid transformation in skeleton animation as it is easier to interpolate and retain the bone hierarchy.

2.3.3 Vertex skinning

Vertex skinning is fundamental for character animation in computer graphics. It generates detailed, high-quality deformations of the characters from a set of control parameters. Vertex skinning is a process of associating each bone with part of the mesh. Each vertex is influenced by one or multiple nearby bones and is transformed via a linear combination of homogeneous transformations of the influenced bones. Figure 2.2 shows an example of bone transformations of the arm bones and figure 2.3 shows an example of influence skinning weights on those bones.

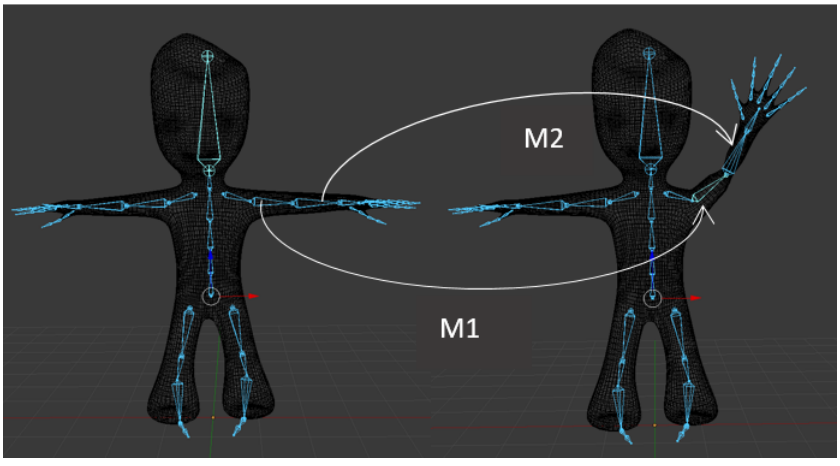


Figure 2.2. Bone transformations of the lower and upper arm bones for one example deformed pose.

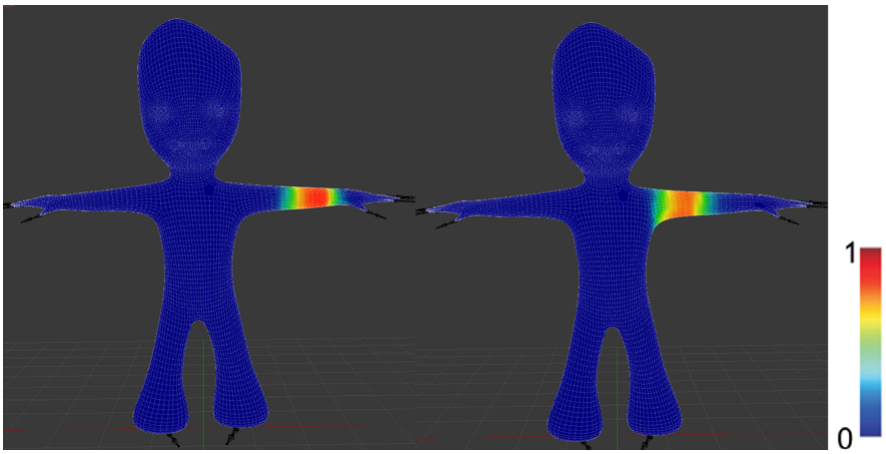


Figure 2.3. Skinning weights corresponding to the lower and upper arm bones.

Normally, vertex skinning requires the following input data:

- *Rest pose polygon mesh.* The mesh is assumed to satisfy constant connectivity, which only the vertex positions will change during deformations. Deformations are considered to take place in a homogeneous space. Thus the mesh is denoted as $v_{1...n} \in \mathbf{R^4}$, where v_i is a vertex vector on the mesh with four coordinates x, y, z, and w. The last coordinate w is often assumed to be equal to 1.

- *Global bone transformations.* These are essentially a set of homogeneous matrices that represent the orientation and position of each bone in the global space, denoted as $T_{1...n} \in \mathbf{R^{3\times4}}$. Bone transformations are used to transfer mesh vertices from their rest pose to an animated pose.

- *Skinning weights.* This describes the influence of one bone on a vertex. It is often convenient to assume that one vertex is assigned with no more than four skinning weights. Skinning weights are denoted as $w_{i,1...i,n} \in \mathbf{R}$, where $w_{i,n}$ describes the amount of influence of a bone n on vertex i. Skinning weights should satisfy the partition of unity where the values are convex and non-zero, i.e. $\sum_{i=1}^{n} w_i = 1, (w_i \geq 0)$.

It is common in vertex skinning to assume the rest pose polygon mesh and the skinning weights are not changing during an animation. The following formula computes the deformed vertex position:

$$v' = \sum_{j=1}^{n} w_{ij} T_j v_i \tag{2.24}$$

To emphasise the fact that the vertex v_i is transformed by a linear blend of transformation matrices, the formula can be highlighted as below:

$$v' = (\sum_{j=1}^{n} w_{ij} T_j) v_i \tag{2.25}$$

The sum of weighted bone transformations T_j illustrates the transformation of the vertex. The formula of transformation blending is the core of vertex skinning techniques. Due to the bone transformations being linearly blended, the method is known as linear blend skinning (LBS).

LBS suffers from the well-known volume loss artifacts, especially when rotation at the joint reaches to approximately 180 degrees. The "candy-wrapper" artifact is revealed when twisting the mesh at the joint, and the "collapsing joint" artifact occurs when the mesh at the joint is significantly bent. Consider the case when linearly blending two rotations R_1 and R_2:

$$R_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, R_2 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

i.e. R_1 is the identity matrix (no rotation) and R_2 represents a rotation about the z-axis by 180 degrees. According to the formula of LBS (formula **??**), an

averaged linear blending of R_1, R_2 can be illustrated as follows:

$$\frac{1}{2} \times \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The resulting matrix shows that the volume that previously existed in the x and y space is being collapsed to 0, which is corresponding to the "candy-wrapper" artifact. The "collapsing joint" artifact happens when joints are severely bent. The problem can be illustrated by the schematic below:

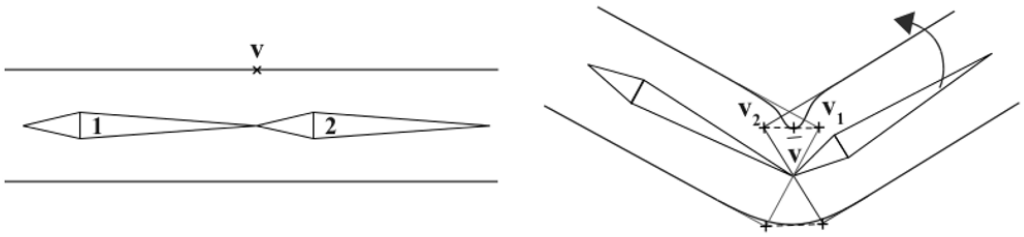


Figure 2.4. Vertex in rest pose (left), and the deformed vertex in an animated pose (right).

As shown in figure 2.4, a vertex with nearly equal weights gets transformed to the closely located point near the joint. Thus the deformed mesh appears as volume shrinkage on the surface.

2.3.4 Keyframes animation

Keyframes animation is a sequence of animation of an articulated character. Each animation keyframe contains information about joint transformations at specific timestamps, including joint rotation, bone translation, and bone scale.

Character animation artists normally create keyframes animation. Artists

define different poses of the skeleton by placing the bones at the desired position with the desired orientation, at the desired timestamp, and mark the pose as a keyframe. Then pre-defined keyframes are interpolated in animation systems to generate a smooth animation cycle. There are three commonly used interpolation methods — step interpolation, linear interpolation, and spline interpolation. This thesis focuses on linear interpolation keyframes animation, which produces intermediate frames between two consecutive keyframes in a linear fashion. The formula below derives the process of finding the parameter for interpolation:

$$f = \frac{animationTime - currentTimestamp}{nextTimestamp - currentTimestamp} \tag{2.26}$$

hence the interpolation of keyframes animation is given by [12]:

$$k = (1 - f)k_1 + fk_2 \tag{2.27}$$

where k_1, k_2 denotes two consecutive keyframes.

Chapter III

Literature Review of Skinning

Skinning is fundamental for character animation in computer graphics. It generates detailed, high-quality deformations of the characters from a set of control parameters. Some skinning methods define these control parameters as rigid transformations associated with the bones. Other skinning methods can determine the control parameters from example poses or physical properties. In this chapter, we present the state of the art of skinning techniques. Broadly, skinning techniques can be divided into three categories: geometric methods, example-based methods, and physics-based methods. Furthermore, geometric methods can be grouped into direct and variational (or indirect) methods, where direct methods compute the deformed skin geometry based on closed-form formulas, and variational methods are based on continuum mechanics, which typically require numerical optimisation [17]. The approach proposed in this thesis falls into the category of direct methods under geometric skinning. Therefore the main focus of this chapter is on this category of skinning techniques.

3.1 Geometric skinning

Geometric methods normally require only one input mesh, and the mesh is deformed according to the transformation of the skeleton. Typically, the new position of a mesh vertex is represented by a weighted sum of homogeneous

transformations of related joints from its rest position to the animated pose. This deformation representation facilitates geometric skinning algorithms to be simple and efficient.

The root of geometric skinning is hard to trace. Catmull [18] first introduced a skeleton-driven rigid skinning technique. Also, Magnenat-Thalmann et al.[19] first presented their thoughts about geometric skinning on human hands. Magnenat-Thalmann's work focused on solving the skeleton motion and surface deformation and introduced two key concepts, bone segments and joints, and a set of parameters to produce a smooth local distortion of the joints. In Magnenat-Thalmann's work, the deformation is achieved from the linear combination of joint-associated deformation. Nowadays, this method is referred to by various names, such as skeleton subspace deformation by Lewis et al.[2], matrix palette skinning, and the more common LBS technique.

LBS is most popular in character animation. Even though it does not always generate realistic deformations, the technique is known for being easy to understand and straightforward to implement. LBS calculates a weighted sum of rigid transformations per vertex in a linear fashion. This linear interpolation of matrices works well when the blended rotational transformations are close. When the joints are twisted significantly, the blended vertices generate the "candy-wrapper" artifact, and when joints are bent by more than 90 degrees the "collapsing joint" artifact appears. Both artifacts cause significant loss of mesh volume because a linear combination of rigid transformations is not guaranteed to be a rigid transformation [20].

The log-matrix skinning [20] technique solves the "candy-wrapper" artifact by blending matrix logarithms, and Magnenat-Thalmann's virtual environment [21] utilises this method. The bones blending technique [22] performs the weighted averaging on the bones instead of vertices to preserve

mesh volume. Although these methods produce adequate results that overcome the volume loss problem of LBS, they both have high computational cost and are considerably slower than LBS. Also, log-matrix skinning introduces new artifacts by choosing a longer trajectory when interpolating rotations [23]. A corrective post-processing method called stretch smooth skinning [24] adds additional stretch operations to both the rest and animated pose to recover the volume. The stretch smooth skinning technique overcomes both the "candy-wrapper" and "collapsing joint" artifacts without extra run-time cost. Kavan et al. [25] enhanced LBS with blend bones. The algorithm automatically selects optimal placement of blend bones and re-computes vertex weights accordingly and performs the classical linear vertex deformations. The resulting distortion overcomes LBS artifacts, and the run-time performance is as fast as LBS as the overhead of evaluating blend bones is negligible.

Another direction of geometric skinning, which is the non-linear technique, can sufficiently solve linear issues. Hejl's work [26] converts bone transformations to (quaternion, translation) pairs and blends the pairs instead of matrices. The linear interpolation of translation vectors is trivial, and the interpolation of quaternions is based on the SLERP technique [8]. However, Hejl's algorithm [26] constraint on the model's rigging is that each vertex can only be influenced by two bones, which is not applicable in vertex skinning pipelines. Spherical blend skinning (SBS) [27] removes the constraints on rigging and is appropriate in the same skinning environment as LBS. The SBS technique interpolates the bone transformations in a non-linear space. SBS overcomes the issue of matrix-based methods and performs the interpolation of rotational transformations on the arc of a unit sphere, which produces a smooth rotation with a constant angular speed. However, the SBS method is computationally expensive (due to it using an elaborate Singular Value

Decomposition scheme) and generates unexpected deformations because of an issue with the selection of the center of rotations (CoRs) when blending the quaternions. Recent work by Disney research extends and improves the SBS technique by pre-calculating optimised CoRs [10] at a per-vertex base. Their algorithm determines an optimised CoR from the sum of similarity between two skinning weights of vertices in normalised skinning weight space. Their method produces consistent and better results than LBS, log-matrix skinning, and SBS.

Skinning with dual quaternions (DQS) [28] leverages the advantages of a dual quaternion that it can represent the rotation of an arbitrary axis along with a translation. The DQS technique linearly blends unit dual quaternions instead of (quaternion, translation) pairs in SBS. The same as LBS, DQS is direct, and the run-time dual quaternion linear blend algorithm (DLB) is a closed form equation [14]. The property results in DQS being as fast as LBS and requires less memory cost. Moreover, DQS is twice as fast as SBS. The results generated from DQS completely overcomes the "candy-wrapper" and "collapsing joint" artifacts.

One major concern is that dual quaternions cannot represent non-rigid transformations. Kavan et al. [9] included a multi-phase indirect method that separates the non-rigid part of the bone transformation, then interpolates the non-rigid transformations after the rigid ones. The same setup appears in the latest Disney's feature film *Frozen*, which is the first production use of DQS to handle rigid and non-rigid bone transformations [29].

Another issue with DQS is the "bulging join" artifact, which is quite visually observable with a 90 degrees rotation or larger. Kim et al. [1] point out the DLB algorithm not only blends unit dual quaternions, but it has the effect of combining the rotation centers. When the rotation center is blended to a single point, a vertex that locates further from the rotation center tends

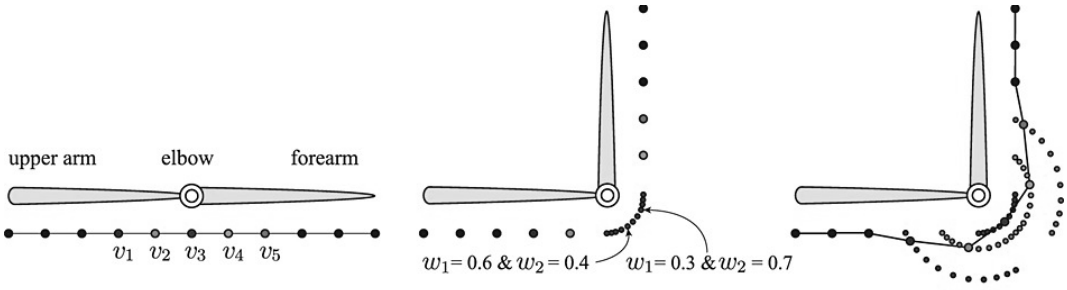


Figure 3.1. The "bulging joint" artifact of DQS. The further the point from the rotation center, the larger the rotation radius the vertex has [1].

to have a larger rotation radius, which causes the deformed mesh at the joint to bulge (see figure 3.1). Their work offers a post-processing bulge-free method to correct the vertex position in its animated space. The proposed method works similarly to the stretch smooth skinning [24] technique. The method pre-computes the distance of each vertex to the nearest bone segment or joint in the rest pose. While animating, if the vertex is detected to have moved further away from the bone segment or joint, they merely project back towards the bone to maintain the original distance. However, Vaillant [30] pointed out that the re-projection introduces a discontinuity in the deformed mesh, and in some cases, the neighbouring vertices may be projected in the opposite direction.

Kavan et al. [31] tackle the artifacts of DQS based on the concept of joint-based deformers. Their research has two main contributions: first, they offer an optimised set of skinning weights for DQS; second, they proposed an indirect closed-form skinning algorithm that non-linearly approximates the elasticity deformers. The resulting deformation has higher quality than the conventional methods DQS and LBS and overcomes the issues thoroughly.

Recently research on indirect skinning methods realises highly plausible skin deformation. Valliant et al. [32] introduced a novel implicit skinning technique that generates skin contacts and muscle bulge effects in real-time

and avoids the volume loss or bulging problem with standard geometric methods. Their skinning approach uses automatically generated implicit surfaces rather than the traditional mesh and applied on conventional geometric skinning techniques. They further extended implicit skinning to produce skin elasticity without the definition of skinning weights [33].

3D software such as Blender [34], Autodesk 3ds Max [35], and Maya's Smooth Skinning [36] provides both LBS and DQS features for animation artists. Furthermore, Maya's Blend Smooth Skinning [5] provides users with the ability to linearly combine DQS and LBS, which is similar to the proposed method of this thesis. Maya's tool requires artists to paint a blend weight map in advance, which generally takes the animator a long time to complete. The painting process is a single direction, iterative, and combines expertise with trial and error. It can be very tedious to achieve the expected quality at all joint regions. Compared to Maya's Blend Smooth Skinning, our method builds on the same animation pipeline setup as the standard DQS and LBS, with no dependencies of extra weight map or change of the original information from the model.

Generally, geometric methods produce acceptable results that satisfy the basic skinning requirements in computer animation systems. Techniques such as DQS and LBS are advantageous because they are extremely fast, which is the reason that they are popular in the game industry and VR applications. However, they are not capable of generating complex and detailed skin deformation, or secondary effects. These deformations are typically handled by advanced skinning techniques such as example-based methods and physics-based methods. In most high-end computer animation applications, advanced procedures are often built on top of DQS and LBS to satisfy specific skinning requirements.

3.2 Example-based skinning

Different from geometric skinning, which specifies the deformed position of a vertex using a short algorithm, example-based approaches interpolate a set of examples of the desired deformation. Example-based skinning is advantageous because of the desired deformation, which is directly sculpted. It gives artists precisely what they expect, and the range of possible deformation is much more extensive than geometric methods because it allows artists to add new shapes as needed to obtain the new desired quality. Among disadvantages, due to the system requiring example poses as input, it is not hard to expect that example-based skinning requires a computation resource proportional to the number of example poses. Especially when considering higher-dimensional poses, such as animating the trunk of an elephant where the number of bones influencing a vertex is more than 10, artists are required to sculpt a large number of example shapes. In practice, example-based skinning is preferred to be implemented on top of geometric skinning to provide skin effects that are limited by the latter one.

The first example-based method is pose space deformation (PSD) proposed by Lewis et al. [2]. PSD can produce fairly convincing results compared to geometric approaches (see figure 3.2). Given many example shapes sculpted from moving the skeleton to different desired poses by 3D artists, the PSD algorithm interpolates these example shapes as a function of pose (or in a pose space) to obtain the final deformation. As an extension of PSD, Sloan et al. [37] presented a method which interpolates the example poses in the abstract space instead of the pose space in PSD. They introduce abstract space and define it as dimensions of global properties of the character model, which adds the character's age and gender to standard properties such as the characters position and orientation. The weighted pose space

Figure 3.2. Comparison on PSD (left) and LBS (right), the skin deformation at challenging areas such as the elbow and the shoulder appears more realistic in PSD [2].

deformation (WPSD) technique [38] was proposed to reduce the number of required poses in PSD, but it increases the computational cost as weights are applied to the standard PSD process. Thus does not apply to real-time interactive applications. Kry et al. [39] presented the EigenSkin technique that significantly reduced the overheads by only using pre-computed principal components of the deformation instead of example poses, and boosted the run-time performance by implementing the algorithm on the GPU. Rhee et al. [40] also presented a GPU shader implementation of WPSD and sped up the performance to 20 times faster than PSD and WPSD. Their algorithm enables the use of PSD and WPSD in interactive applications. Wang et al. [41] proposed a real-time example-based solution which uses a rotational regression model to capture deformations such as muscle bulging, and twisting at challenging areas such as the shoulders. Their method generates more accurate skin deformation than LBS and is almost as fast. Loper et al. [42] proposed the skinned multi-person linear (SMPL) model, which is fully compatible with existing graphics pipelines because it uses standard skinning methods to transform the deformed shapes (with influences on the shapes)

into the desired pose. The resulting deformation realistically represents any body shape in any pose.

Another direction of example-based skinning methods is the weight enveloping technique. The multi-weight enveloping (MWE) [43] technique estimates various skinning weights from a set of example poses. The method advocates a linear model with 12 skinning weights at each vertex of an associated bone. By adding more skinning weights, the MWE method can avoid the "candy-wrapper" artifact, but with the cost of extra complexity during vertex skinning and 12 times as many skinning weights than LBS and DQS.

Example-based methods can handle complex skin deformations that geometric methods are not able to. Pyun et al. [44] cloned facial expressions with an example-based approach. Park et al. [45] introduced a data-driven method to dynamically synthesis detailed skin deformation. Shi et al. [46] proposed a subspace-based model to approximate the non-linear elasticity effects from a set of physical behaviours of a mesh. Their skinning technique can produce the jiggling of fatty tissues and is suitable for real-time applications. Disney introduced the use of PSD for facial animation [47] [48]. Recently, Schumacher et al. [49] used linear interpolation of example poses to simulate art-directable deformable materials. Zurdo et al. [50] used WPSD to add a dynamic wrinkle detail to a low-resolution cloth model. In another direction, Le et al. [51] automatically generated LBS models with skeletons based on example-based rigging.

Even though example-based techniques are easy to implement and produce realistic results, which is missing from geometric methods, it requires a large amount of effort from 3D artists to manually create a wide range of examples in advance, or from a complex physics simulation system when real characters are not applicable. Furthermore, to generate dynamic physics phenomena, such as skin contact deformation due to a collision, the artist

must configure deformed poses for each frame. It is time-consuming and not applicable in reality. Therefore, physics-based methods provide artists with a better alternative for more advanced skin deformations.

3.3 Physics-based skinning

While example-based skinning can provide effects such as muscle bulging and skin contact by requiring examples from the artist, it is a tedious process. Therefore, physics-based skinning serves as an alternative that employs dynamic physics phenomena into the skinning process. The obtained effect is highly realistic.

Unlike skeleton-based skinning methods, which aim to move the skin according to the kinematics of the underlying skeleton, the primary goal of physics-based skinning is to simulate all the secondary effects that are missing from geometric methods. Secondary effects enrich the visual experience of animation and are essential for production films. Professional tools also include packages for physics-based simulation, such as Maya Muscle [52] and Weta Digital's Tissue System [53].

Some research studies focus on producing believable secondary effects (see figure 3.3). McAdams et al. [4] proposed an algorithm based on a discretisation of corotational elasticity for skinning with contacts and collisions. Liu et al. [3] present a physics-based framework for simulating a soft body based on skeletons. Moreover, Rmillard et al. [54] proposed a two-way coupled model that combines the high resolution thin shells with the coarse finite element lattices for simulating high-resolution surface wrinkle deformations. Li et al. [55] focus on stretching and sliding the thin hyperelastic skin by modeling an Eulerian representation of skin.

Recent research has achieved exciting results. Kim et al. [56] proposed a method which combines the advantages of data-driven methods and

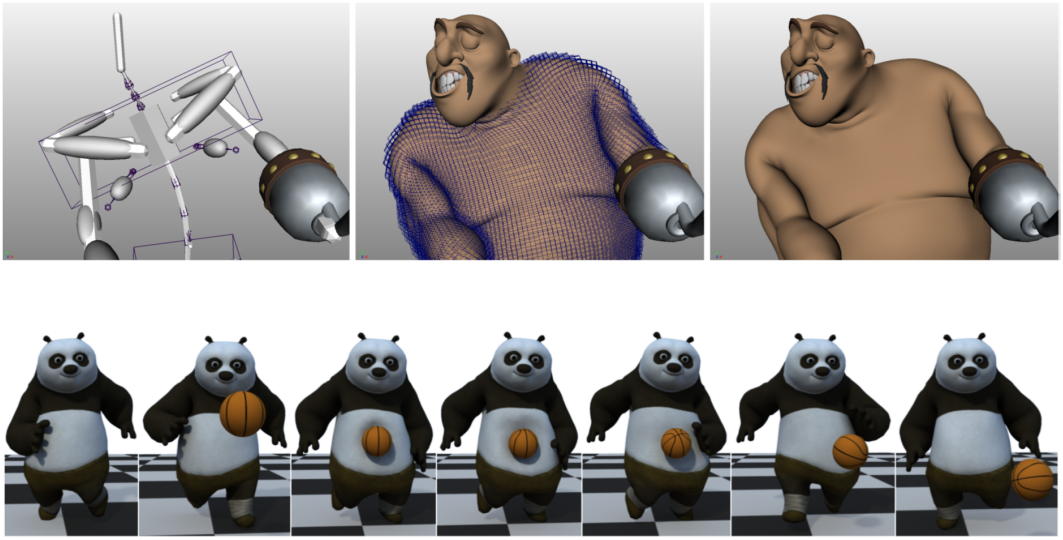


Figure 3.3. (Top) skeleton-based simulation of soft body [3] and (bottom) elasticity skinning with contacts [4].

physics-based methods. They used a multi-layered model for simulation, with the inner layer driven by a statistical body model and the external layer driven by physics simulation. Their method achieves realistic results and reacts to external forces, and supports the re-targeting of physical properties from the character model. Luo et al. [57] proposed a method that simulates non-linear deformable objects in a spatial reduction framework by using overlapping quadratic domains. Their proposed method incorporates high-order degrees of freedom to ensure simulation quality. Zhou et al. [58] recently disclosed a real-time motion simulation method based on pre-computation training data, but the simulation is limited to hair-object collisions only.

Physics-based skinning techniques are the best choice to produce high-resolution skinning effects, but setting up a character with an anatomical model and appropriate material properties is costly. Hence, it is hard for physics-based approaches to achieve real-time capability on complex scenes with external forces as the complexity of computing for each frame is reasonably expensive.

Chapter IV

Proposed Algorithm: Theoretical and Implementation Aspects

For this dissertation, the intention is to implement a character animation application that deforms the character with an improved dual quaternion-based vertex skinning algorithm. The algorithm will reduce the "bulging joint" artifact of standard dual quaternion skinning (DQS) by bringing back a specific amount of linear blend skinning (LBS) features while reducing the influence of DQS. The resulting deformation corresponds to a mixture of DQS and LBS. The proposed method would be suitable for use in interactive applications such as VR and video games, which require skinning in real-time. Using Kavan's paper [9] with the published GPU implementation of DQS [59] as a base point, the standard DQS technique will be reproduced by this thesis to generate DQS skinning effects, especially the "bulging joint" artifact as ground truth.

This chapter presents the design and implementation of the character animation application, and explains the GPU implementation detail of the proposed vertex skinning algorithm, which is the core of this thesis. Section 4.1 explains the theoretical aspects of the proposed method. Sections 4.2 and 4.3 present the technical specifications and high-level design of our character animation system. Section 4.4 discusses the implementation detail of the animation system at class-level. Details of the proposed vertex skinning

algorithm are discussed in section 4.5. Variations on the DQS implementation approaches and the choices made in this implementation are discussed in section 4.5.1. Section 4.6 presents the user interface of the implemented application, and a discussion on the mesh model validity and the issue of integrating Kavan's GPU implementation [59] is given in section 4.7.

4.1 Combining DQS and LBS

The proposed method will perform the standard LBS to obtain the LBS specific deformation, and replace a certain amount of the DQS deformation with it. According to the concept of linear combination [60] of vector $x_1, ..., x_n$ in Euclidean geometry:

$$\sum_{i=1}^{n} a_i x_i = a_1 x_1 + a_2 x_2 + ... + a_n x_n \tag{4.1}$$

where the coefficients a_i are scalars. The linear combination of a vertex vector, which is transformed from DQS and LBS, is written as:

$$v' = a_1 v_d + a_2 v_l = a_1 (T_d v) + a_2 (T_l v) \tag{4.2}$$

where v' is the linear combined vertex vector, T_d and T_l represents the transformation matrix computed from DQS and LBS, v_d and v_l corresponds to the vertex transformed with DQS and LBS techniques, and v is the vertex vector in rest pose. When the sum of the coefficients is 1, that is, $\sum_{i=1}^{n} a_i = 1$, v' represents an affine combination of v_d and v_l. Therefore, formula 4.2 becomes:

$$v' = a(T_d v) + (1 - a)(T_l v) \tag{4.3}$$

Due to the affine combinations commuting with any affine transformation T in the sense that:

$$T\sum_{i=1}^{n} a_i x_i = \sum_{i=1}^{n} a_i T x_i \tag{4.4}$$

Formula 4.3 can be re-written as:

$$v' = [a(T_d) + (1-a)(T_l)]v \tag{4.5}$$

The derived formulas (formula 4.3 and 4.5) indicates two possible approaches for our implementation:

1. Linearly interpolates the DQS and LBS transformations, and apply the resulting matrix on the vertices.

2. Linearly interpolates the DQS transformed vertices and the LBS transformed ones.

Both approaches require an interpolation parameter, which is the introduced "deform factor". The "deform factor" will be adjustable (range from 0 to 1) in real-time via a UI component, where 1 corresponds to the standard DQS technique, and 0 corresponds to the standard LBS technique.

4.2 Technical specifications

The work will be implemented on an Intel Core i7-6700 3.6 GHz processor Windows 10 (64-bit) machine with 8 GB ram and an Nvidia GeForce GTX965M (Support OpenGL Version 4.5) graphics card.

The skinning results will need to be visualised for observation and evaluation. Therefore, an OpenGL application will be built from scratch, which supports model import and keyframes animation. The application will also contain elementary graphics scene objects such as a camera and a light.

GLFW [61] and GLEW [62] libraries will be used for the implementation of the application.

Rigged character models are obtained from 3D model providers such as Mixamo [63] and Sketchfab [64] under a free licence. Model files will be imported to Blender software [65] for validation before being exported to the .fbx files. Then the models will be imported to the application using the ASSIMP library [66]. The ASSIMP library will also parse the model and store the data in its structure. Therefore the application will obtain mesh definitions, bones, materials, and animations and be stored in OpenGL's data structure. Obtained keyframes animation will need to be interpolated for smooth character animation.

The implementation of vertex skinning algorithms will target the GPU, thus the proposed improved skinning algorithm will apply to each vertex via a vertex shader. The modern OpenGL supports GPU programming with shaders, and shader programs are written in OpenGL Shading Language (GLSL). The application will create a modern OpenGL context with version 4.5, which is the latest version supported by the graphics card used for this research. An HLSL version of the vertex shader is published by Kavan [59], which only runs under Microsoft's Direct3D environment, and it will require a re-implementation in GLSL to be suitable for use in this application.

Calculation of dual quaternions and matrices will frequently occur in the implementation, and the GLM math library (OpenGL Mathematics) [67] will be used for calculations of relevant math models. The GLM library includes functions for constructing quaternions and dual quaternions given transformation matrices, and provides functions for the arithmetic operations which are discussed in section 2.1 and section 2.2.

Kavan's implementation of DQS required dual quaternions to be pre-calculated on the CPU and passed into the GPU. Therefore, the following

elements will be required to reproduce the standard DQS:

- The rest pose mesh of the model. The definition of a single mesh consists of vertex positions, normal vectors, and texture coordinates.

- A collection of dual quaternion bone transformations.

- A list of bone indices for each vertex, which indicates the influencing bones of that vertex.

- A list of assigned skinning weights for each vertex, which corresponds to the list of bone indices and determines how much a particular bone influences the vertex.

In order to perform the proposed skinning algorithm, the standard LBS will be implemented with most of the required elements as with the DQS listed above. The difference is that instead of passing an array of dual quaternions to the GPU, LBS requires an array of bone transformation matrices. Linearly combining of DQS with LBS will require the "deform factor", and it will be directly injected into the GPU via the uniform variable supported in the model OpenGL. The "deform factor" will need to be adjustable via the UI component, and the ImGui library [68] will be integrated with OpenGL to draw a slider bar ranging from 0 to 1 for the "deform factor" and allows users to adjust the value in real-time. For this research, only a vertex shader and a fragment shader will be implemented on the GPU, where the vertex shader computes the transformation for each vertex and the fragment shader computes vertex colours.

The libraries and tools will be used for this implementation are listed here:

- GLFW (v3.2.1): A multi-platform library for OpenGL. It provides APIs for creating windows, contexts and surfaces, and receiving inputs and events.

- GLEW (v2.1.0): A cross-platform C/C++ extension loading library, provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in the single header file.

- ASSIMP (v4.1.0): A popular C++ model importing library.

- Blender (v2.79): An open source 3D graphics tool. It is free and widely used for creating 3D assets, animations, movies, even video games.

- GLM (v0.9.9): A header-only math library, which provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL. Moreover, it contains various extensions such as matrices transformations, quaternions, and dual quaternions.

- Stb_image.h (v2.19) [69]: A single header image loader library that can load various image file formats.

- ImGui (v1.6.6): A graphical user experience library that can be used to draw UI components on the screen, regardless of types of graphics API.

- Visual Studio Community 2017: An IDE platform by Microsoft.

4.3 High-level design

The schematic overview of the system structure is indicated in figure 4.1. Here a character model file is exported to the .fbx file from Blender. The

model with textures is imported through ASSIMP to the main OpenGL application. The application loads the model file at start-up time and extracts data from ASSIMP's data structure. Extracted model data will be parsed and stored in OpenGL's data structure. Once the processing of the model is finished, the animation data will be extracted and the interpolation step will be performed to interpolate keyframes animation based on the animation time. The pre-computation of dual quaternions and bone transformation matrices is performed during the interpolation process. After this pre-computation step is finished, both dual quaternions and bone transformation matrices are passed into the GPU with vertex-level data for vertex skinning. The GPU performs the proposed improved vertex skinning algorithm, according to the state of the "deform factor", and applies the final transformation on each vertex.

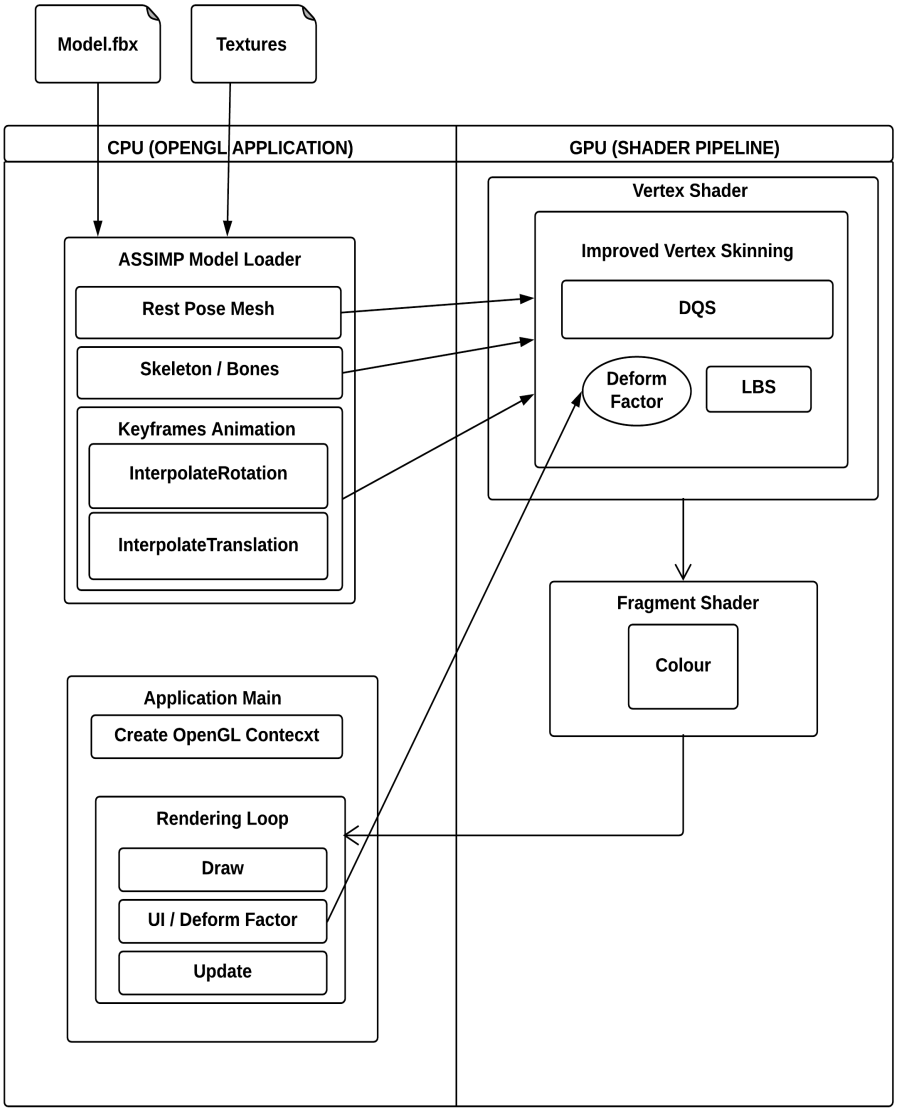


Figure 4.1. A high-level overview of the system

With regard to the individual components of the application, the "ASSIMP Model Loader" component will perform three main steps once the model file is imported: first, store the mesh definitions; second, construct the storage for the hierarchy of bones; and third, component "Keyframes Animation" will execute the methods "InterpolateRotation" and "InterpolateTranslation", which will compute the interpolated rotation matrix and translation matrix, and create the bone transformation matrices and corresponding dual quaternions. The bone transformation and dual quaternion, along with the mesh definition, will be passed to the shader pipeline on the GPU. The "Application Main" component will create the OpenGL context, and it will also have a "Rendering Loop" component to draw and update each frame. The UI component for interactive controls of the "deform factor" will be integrated to the "Rendering Loop". The "Application Main" will directly insert the current state of the "deform factor" to the shader program on the GPU.

On the GPU, the shader pipeline will consist of two shader programs. Once the vertex shader receives data from the CPU, a linear combination of the standard DQS and a specific amount of LBS influences determined by the value of the "deform factor" will perform on each vertex. Finally, the fragment shader will compute the pixel colour according to the transformed normal vectors, and the lighting setup before the deformed model renders on the screen.

4.4 Class-level details

Figure 4.2 presents a class diagram of the OpenGL application. The boilerplate code such as the ShaderLoader class is omitted. The Camera class and Lamp class are also omitted as their main methods and fields are irrelevant to the implementation of skeleton animation. Methods which are

used for debugging the application are also omitted. In figure 4.2 all classes use the ShaderLoader class. The main function of the ShaderLoader class is to load and compile the shader files (vertex shader and fragment shader). This class handles reading and compiling GLSL shader programs, checking for errors and creating the executable shader programs. Besides, this class provides support for passing data to the GPU via OpenGL uniform variables. Uniform variables are OpenGL buffer objects for passing data to the shader pipeline. The ShaderLoader class is used in the method Draw() in the Model class to pass model data to the vertex shader before drawing the model and rendering all its meshes.

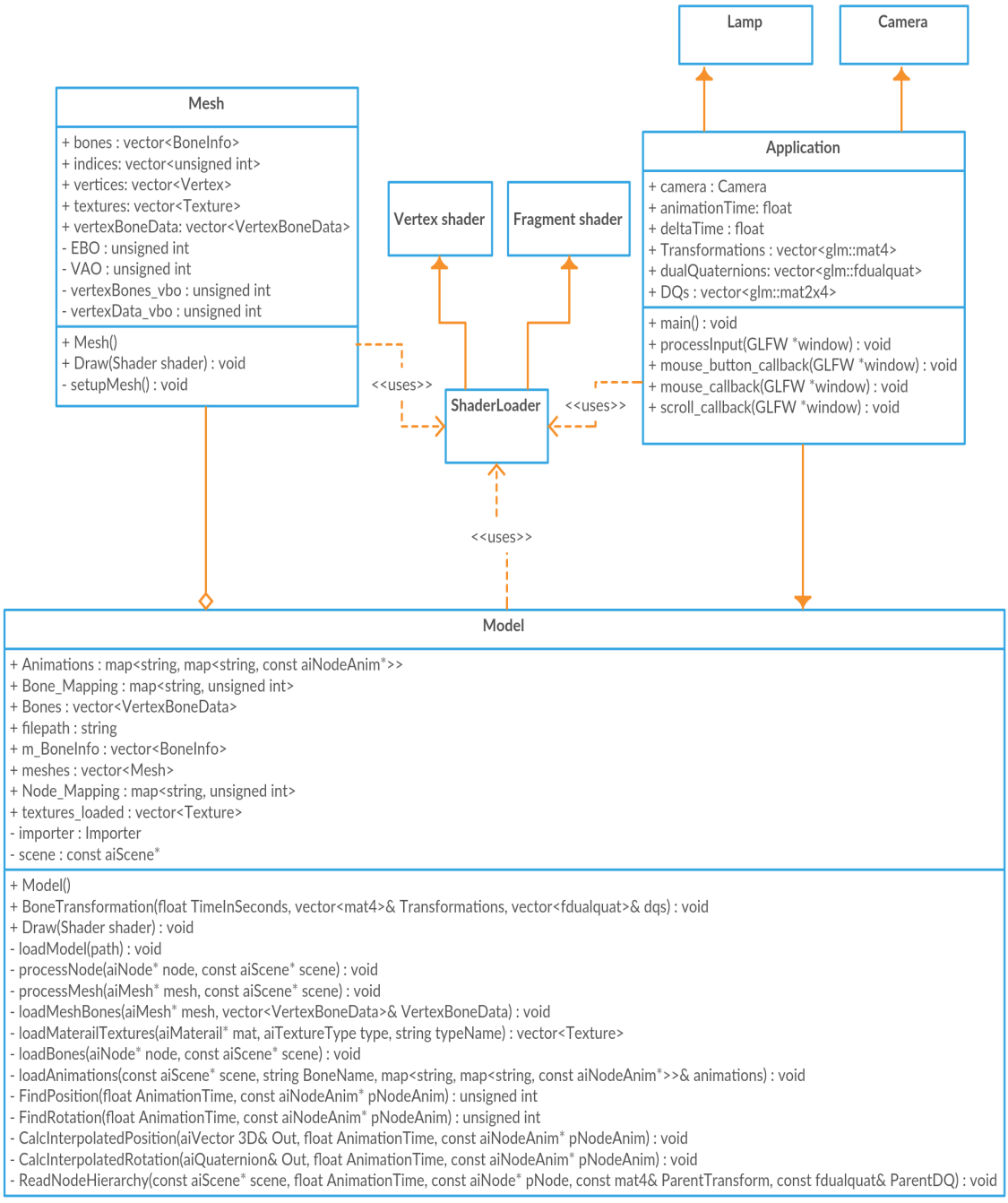


Figure 4.2. A class diagram of the system

The important fields of the Application class are:

- animationTime: It is used in keyframes interpolation. This variable

represents the time between the current frame and the last frame and stores the current position at the time of the animation.

- Transformations: A vector for storing the bone transformation matrices.

- dualQuaternions: A vector of the GLM dual quaternion type. Dual quaternions are stored in (rotation quaternion, translation quaternion) format in GLM.

- DQs: A vector for storing dual quaternions in the matrix form, as GLSL does not support the GLM dual quaternion type.

The method main() of the Application class contains the OpenGL context, along with the rendering loop. In general, the Application class first creates the OpenGL context via GLFW configurations, then constructs the model from the specified directory and performs vertex skinning, and finally renders and updates the scene in the loop. The rendering loop terminates when the user closes the application window. In the rendering loop, the critical actions performed are:

- Calculate animationTime using the timestamps of the start frame and current frame.

- Activate the GPU shader pipeline (or shader objects).

- Initialise the "model", "view", "projection" matrices, and send them to the currently activated vertex shader.

- Call the Model class method "BoneTransform()" to calculate bone transformations of the current animationTime. The result is stored

in the *Transformations* vector in the form of a matrix, and the *dualQuaternions* vector in the form of a dual quaternion.

- Send the bone transformations and dual quaternions to the activated vertex shader program.

- Initialise the OpenGL uniform variables and pass them to the activated vertex shader program.

- Initialise the UI components and define the "deform factor", and display the UI component on the screen.

- Call the Model method "Draw()" to render the animating model on the screen.

The actions are performed in a specific sequence as OpenGL is essentially a state machine and the shader program only executes if activated. In this implementation, the uniform variables initialised and used are listed:

- "Model" , "View", "Projection" matrices (4×4 matrix).

- The vector of bone transformation matrices (3×4 matrix).

- The vector of dual quaternions matrices (2×4 matrix).

- The "deform factor" (float).

The Mesh class provides a storage area for the vertex level information associated with each mesh, and passes the vertex data to the GPU shader programs via several OpenGL shader functions. The constituents of the Mesh class are:

- Storage for the required data of a mesh, including collections (vectors) of vertices, indices, textures, bone transformations, associated bones, and skinning weights.

- The constructor prepares all the required data of a mesh, then uses the setupMesh() method to set the vertex buffers and vertex attribute pointers.

- Vertex: A structure of a vertex object represents a vertex on the mesh, which includes all the information at the vertex level. A vertex object contains the position coordinates, the normal vector, and the texture coordinates.

- Texture: A structure of a texture object represents one texture associated with the mesh, which includes information about the texture file path, the index of the texture, and the type of the texture.

- VertexBoneData: A structure of a vertex's associated bones and corresponding skinning weights, which contains an array of bone indices and another array of weights. By default a vertex has enough storage for four bones, hence the size of both arrays is limited to four. The structure has a method AddBoneData(), which is used to add a new bone index and the weight of the vertex. The method finds available slots in the bone index array and weights array when a new pair of (bone index, weight) is found. This method is used in the method loadMeshBones() of the Model class when creating the vertex-bone binding of a mesh.

- BoneInfo: The structure of the bone information of a mesh, which includes a reference to the parent, an offset matrix and a final transformation in the form of a matrix and dual quaternion. The reference to parent bone is useful for constructing the node hierarchy. The offset matrix transforms the mesh to its local bone space, and the final transformation returns the mesh to skeleton space and determines the final position and orientation of mesh vertices. The bone information stored

here corresponds to the requirement of the process of transferring a mesh vertex that is attached to a bone, which was discussed in section 2.3.2.

- setupMesh(): Used to initialise and configure the vertex array and vertex buffer objects (VAO and VBO). During initialisation, the method creates a VAO, a VBO to store vertex data, a VBO to store associated bones and skinning weights, and an element buffer object (EBO) to store vertex indices. The combination of the two VBOs represents all the vertex-level information the GPU shader pipeline requires. The data is interpreted as vertex attributes and is stored in the VAO with a specified layout. The detailed interpretation is made by the OpenGL function call glVertexAttribPointer(), which configures five vertex attributes with the order: positions, normals, texture coordinates, associated bones, and skinning weights.

- Draw(): Used to draw a single mesh object on the screen. This method binds the configured VAO to the vertex shader program and binds activated textures to the fragment shader program. It is used in the Draw() method in the Model class to render all meshes of a model. During every draw call, this method calls the OpenGL function glDrawElements() to draw triangle elements with the vertex indices stored in the EBO, thus rendering a single mesh.

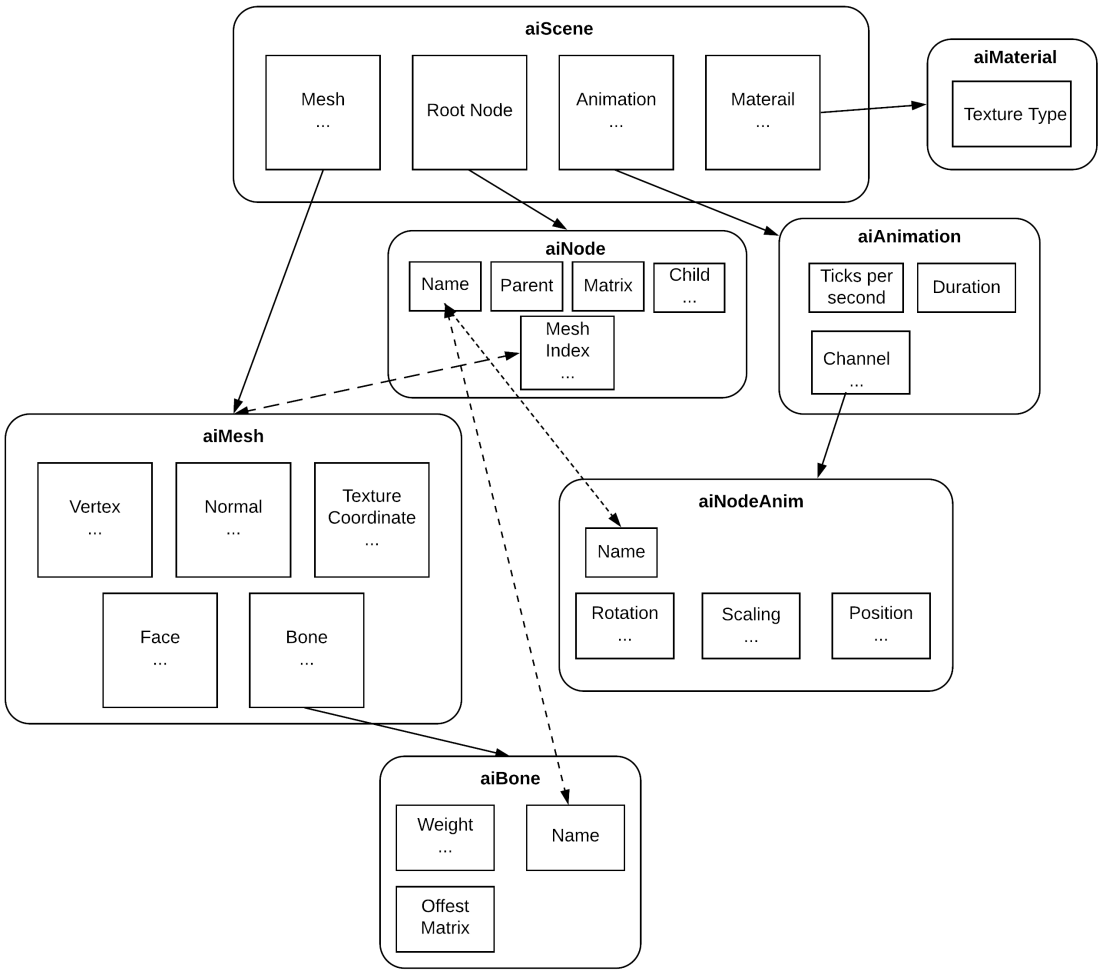


Figure 4.3. A procedure overview of importing models with ASSIMP, the dotted arrows indicates the correspondence of bones and nodes through names.

The Model class represents a model in its entirety. The class contains most of the vital functions of character animation. In general, the Model class performs two steps: first, load the entire model with ASSIMP; and second, process the model data and interpolate animation.

An overview of the implementation procedure of the loading model with ASSIMP is illustrated in figure 4.3. In ASSIMP, the entire model is stored in the **aiScene** object, with a reference to the **RootNode**, which is treated as a

pointer to the **aiScene** object. ASSIMP collects the entities in the model file in a tree, which is intuitive for implementing a node hierarchy in OpenGL. ASSIMP treats nodes equivalent to bones, and correspondence between bones and nodes is through names. The node hierarchy can therefore abstract the bone hierarchy of the skeleton. Once the hierarchy is constructed, all the entities that are relevant to identify the surface geometry of the model (such as the meshes, faces, bones transformations, animations and materials) can be retrieved from the **aiScene** object in a recursive fashion.

ASSIMP provides post-processing commands, which can be specified when loading the model. Table 4.1 shows a list of post-processing commands this implementation employs.

ASSIMP post-processing commands	**Command features**
aiProcess_Triangulate	Triangulates all faces of all meshes
aiProcess_FlipUVs	Flips all UV coordinates along the y-axis and adjusts material settings and bitangents accordingly
aiProcess_GenSmoothNormals	Generates smooth normals for all vertices in the mesh
aiProcess_LimitBoneWeights	Limits the number of bones simultaneously affecting a single vertex to a maximum value
aiProcess_CalcTangentSpace	Calculates the tangents and bitangents for the imported meshes
aiProcessPreset_TargetRealtime_MaxQuality	Optimise the data for real-time rendering

Table 4.1. The list of implemented ASSIMP post-processing commands

Once successfully loads the model, the data is translated and stored in the OpenGL data structure for further processing. The critical fields of the

Model class are shown in figure 4.2 and listed below:

- Textures_loaded: A vector which stores the Texture objects that are defined in the Mesh class. This field contains all the loaded textures of the model.

- Meshes: A vector which stores instances of the Mesh class. A model can have multiple meshes, which are all stored in this field to render the complete mesh model.

- Directory: A string field that specifies the directory of the .fbx model file and textures.

- m_BoneInfo: A vector which stores the BoneInfo objects that are defined in the Mesh class. This field contains the complete bone information of the mesh model.

- Bones: A vector which stores the VertexBoneData objects that are defined in the Mesh class. This field contains the vertex-bone bindings of all vertices.

- Bone_Mapping: A map which assigns an index to a bone name. If a new bone is added, its index is the increment of the current number of bones in the map. The bone index is used for storing BoneInfo objects in the m_BoneInfo vector.

- Animations: A nested map that is used to store the animation of a node (or bone). The key for the first map is animation_name, and the key for the nested map is bone_name.

The important methods of the Model class are:

- The constructor is used to load the model with supported ASSIMP extensions from the .fbx model file.

- Draw() method, which calls the Mesh class's Draw() method to render the entire mesh model on the screen. Without any bone transformation applied, this method renders the rest pose mesh model.

- loadModel() method, which loads a model with specified ASSIMP post-processing commands and is used in the constructor.

- processNode() method, which processes **aiNode** objects in a recursive fashion to build the node hierarchy (or the abstract bone hierarchy) and is used in the loadModel() method. At each node, this method extracts the **aiMesh** objects and calls the processMesh() method to process each mesh located at the node and repeats this process on its children nodes if any. Each processed mesh is then pushed to the vector of meshes (the field *meshes*).

- processMesh() method processes each individual **aiMesh** object. This method is used in the processNode() method. It walks through each of the mesh's vertex, retrieves the vertex positions, normals, and texture coordinates and creates a vertex object (defined in the Mesh class) and pushes the vertex object to the vector of vertices (the field *vertices*). The method retrieves vertex indices from the **aiFaces** of the mesh, and pushes vertex indices to the indices vector (the field *indices*). This method also calls the loadMaterialTextures() and loadMeshBones() method to retrieve textures and the bone information of the mesh.

- loadMeshBones() method, which retrieves the bone information of the

mesh. It is used in the processMesh() method. For a given bone of a mesh, this method first looks up the Bone_Mapping map to check if the bone is already processed. If not, it stores the bone name with a new index. Then it extracts the bone's offset matrix and stores the matrix in the m_BoneInfo vector. Then it retrieves the index of influenced vertices of the current bone, and calls the AddBoneData() method of the VertexBoneData object (defined in the Mesh class) to store the associated skinning weights in the VertexBoneData vector. The resulting VertexBoneData vector contains the vertex-bone bindings (index of associated bones and associating skinning weights) of all vertices.

- ReadNodeHierarchy() method, which retrieves the animation from each bone in the node hierarchy. For a given bone, it determines the positions and orientations defined by keyframes animation. The node transformation in the bone space is calculated from left-multiplying a translation matrix to a rotation matrix and combines with the transformation matrices of all the parents of the current node to result in a global transformation matrix. The global transformation matrix is then combined with the bone offset matrix to result in the final bone transformation matrix. Each final bone transformation matrix is converted to a unit dual quaternion. Both the matrices and unit dual quaternions are stored in the application and passed to the shader pipeline once the vertex shader program is activated.

- Further private access methods are provided to perform the following:

  - Find the index of a key rotation and translation in the list of those stored in the **aiNodeAnim** object, which is immediately before the current animation time.

– Interpolate the obtained key quaternions (rotations) and translation vectors of a specified animations channel based on the current animation time.

4.5 Skinning algorithm implementation details

In the previous section, we explained the implementation on the CPU side, and this section covers the implementation that takes place on the GPU shader pipeline. We discuss three DQS implementation approaches and the choice made for this thesis, and details of our proposed run-time vertex skinning algorithm.

The implementation of the proposed vertex skinning algorithm is in the vertex shader. The bone transformations data we obtained previously from the CPU is the input of the vertex shader. Other inputs correspond to the vertex attributes that are specified in the Mesh class. The vertex shader receives one array of bone transformation matrices and another array of dual quaternion transformations. Since the vertex shader computes one vertex at a time, the transformation data corresponding to the current processing vertex is obtained from the list of bone indices of the associated vertex. The vertex shader contains two functions. The Main function computes the weighted sum of bone transformations according to the LBS formula (formula 2.25) and the DLB formula (formula 2.23), and performs the proposed skinning algorithm according to formula 4.1. The DQtoMat() function converts a unit dual quaternion to a 3×4 matrix following the matrix form of the dual quaternion presented in section 2.2. The output of the vertex shader is passed to the fragment shader in the shader pipeline.

The pseudo code of the vertex shader is shown below:

Input: Vertex attributes:

Position, normal, texture coordinates, an array of four bone indices and an array of four skinning weights

Uniform: Model, View, Projection matrices

Uniform: Array of bone transformation matrices

Uniform: Array of dual quaternion transformations

Uniform: The "deform factor"

Output: Normal, fragment position, texture coordinates

Function:

```
1 DQtoMat() //convert a unit dual quaternion to a matrix
2 Main() //perform vertex skinning
```

4.5.1 DQS implementation approach

The original work of Kavan [9] provided with a vertex shader implementation of DQS [59], which contains three different approaches: the basic approach, the per-vertex antipodality handling approach, and the optimised approach.

The basic DQS calculates the linear interpolation of dual quaternions based on the DLB algorithm described in section 2.2 and converts the obtained dual quaternion to a transformation matrix. The per-vertex antipodality handling DQS calculates a transformation matrix, the same as the basic DQS approach with additional antipodality handling, which solves the longer rotation path issue caused by the "double cover" property of quaternions. Note the process of antipodality handling performed in the vertex shader is before dual quaternion interpolation. The optimised DQS requires the antipodality handling process to perform on the CPU. Furthermore, this approach avoids the dual quaternion to matrix conversion step in the vertex shader and directly calculates the transformed position of the vertex from

the rotation and translation components of the dual quaternion.

We experimented with all three DQS approaches, and it appears that the basic DQS generates undesired problems with the deformed mesh when the joint rotation is 180 degrees (see figure 5.1). Hence, the basic DQS implementation is not robust, and antipodality handling is necessary. The choice between the antipodality handling and optimised DQS is essentially the necessity of converting unit dual quaternions to matrices.

Recalling the two possible approaches of our implementation described in section 4.1, the first approach requires the unit dual quaternion to convert to a transformation matrix in order to combine with the LBS transformation matrix. The second approach omits this conversion and transforms vertices directly with a blended unit dual quaternion, where Kavan's optimised DQS implementation can be applied. However, the second approach results in each vertex being transformed twice (first by a dual quaternion transformation, second by an LBS transformation matrix). Besides this double transformation overhead, this approach leads to a linear interpolation of vertex normal vectors, and undesired lighting artifacts are likely to occur (see the artifact of linearly interpolated normals in Appendix A).

Even though the linear interpolation of vertex position vectors is trivial, the normal vectors must be transformed by the inverse transpose of the transformation matrix. Therefore, in the case that our implementation utilises the optimised DQS, the unit dual quaternion to matrix conversion is unavoidable as the correct transformation of normal vectors is required. For this reason, the implementation of our proposed run-time algorithm is based on the second approach described in section 4.1, which is to combine the DQS and LBS transformation matrices linearly. Our implementation of DQS adopts Kavan's per-vertex antipodality handling approach with the dual quaternion to matrix conversion. The proposed vertex skinning algorithm, along

with antipodality handling and the dual quaternion to matrix conversion, is performed in the vertex shader. The next section presents the run-time algorithm of the proposed vertex skinning technique.

4.5.2 Linear combination of DQS and LBS

The implementation of the run-time algorithm is based on the theory of linear combination of vectors, which has been described in section 3.1. The proposed algorithm for the linear combination of DQS and LBS contributions via the "deform factor" is as follows:

Algorithm 1: Vertex skinning algorithm

Input: unit dual quaternions $\hat{q_1}, ..., \hat{q_p}$
bone transformation matrices $T_1, ..., T_p$ $(T_i \in R^{3\times4})$
vertex position v and normal v_n
joint indices $j_1, ..., j_n$ and convex weights $w_1, ..., w_n$
"deform factor" f
Output: transformed vertex position v' and normal v'_n

1 $\tilde{T} = w_1 T_{j_1} + ... + w_n T_{j_n}$
2 $\hat{b} = w_1 \hat{q}_{j_1} \oplus ... \oplus w_n \hat{q}_{j_n}$
3 **where:** $\hat{q_a} \oplus \hat{q_b} = \begin{cases} \hat{q_a} + \hat{q_b}, \textbf{if } \hat{q_a} \cdot \hat{q_b} \geq 0 \\ \hat{q_a} - \hat{q_b}, \textbf{otherwise} \end{cases}$
4 **normalise and convert** $\hat{b}$ **to matrix** Q $(Q \in R^{3\times4})$
5 **linear interpolation:** $M = (1 - f) * \tilde{T} + f * Q$
6 $v' = Mv$ //**where** v **has form** $v = (v_0, v_1, v_2, 1)$
7 $v'_n = (M^{-1})^T v_n$ //**where** v_n **has form** $v = (v_{n,0}, v_{n,1}, v_{n,2}, 0)$

The linear interpolation of unit dual quaternions uses the $\oplus$ operator (line 2) to perform the antipodality handling. The $\oplus$ flips the sign of the dot product of two unit dual quaternions, which is discussed in section 2.1. Note that at line 7, the normal vector is transformed by the inverse transpose of the final transformation matrix.

4.6 User Interface

The animation application implemented for this thesis comprises several graphical user interface elements developed using the ImGui API. Figure 4.4 shows that the application window holds a list of controls implemented for examining the resulting skin deformation of the proposed algorithm. The primary controls are as follows:

- Wireframe Mode: A checkbox component which renders the model in the wireframe mode (see figure 5.6 and figure 5.7).

- Show Joints: A checkbox component which displays the model skeleton (see figure 5.6 and figure 5.7).

- Show Weight Distribution: A checkbox component which shows the weight distribution of the entire mesh model (see figure 5.3 and figure 5.4).

- Pause: A checkbox component which pauses the keyframes animation.

- Ratio on DQS: A floating slider component which adjusts the "deform factor". The slider varies from 0 to 1, and the value on the slider implicates the amount the DQS contribution.

- Application average framerate: A text component displays the averaged framerate of the animation.

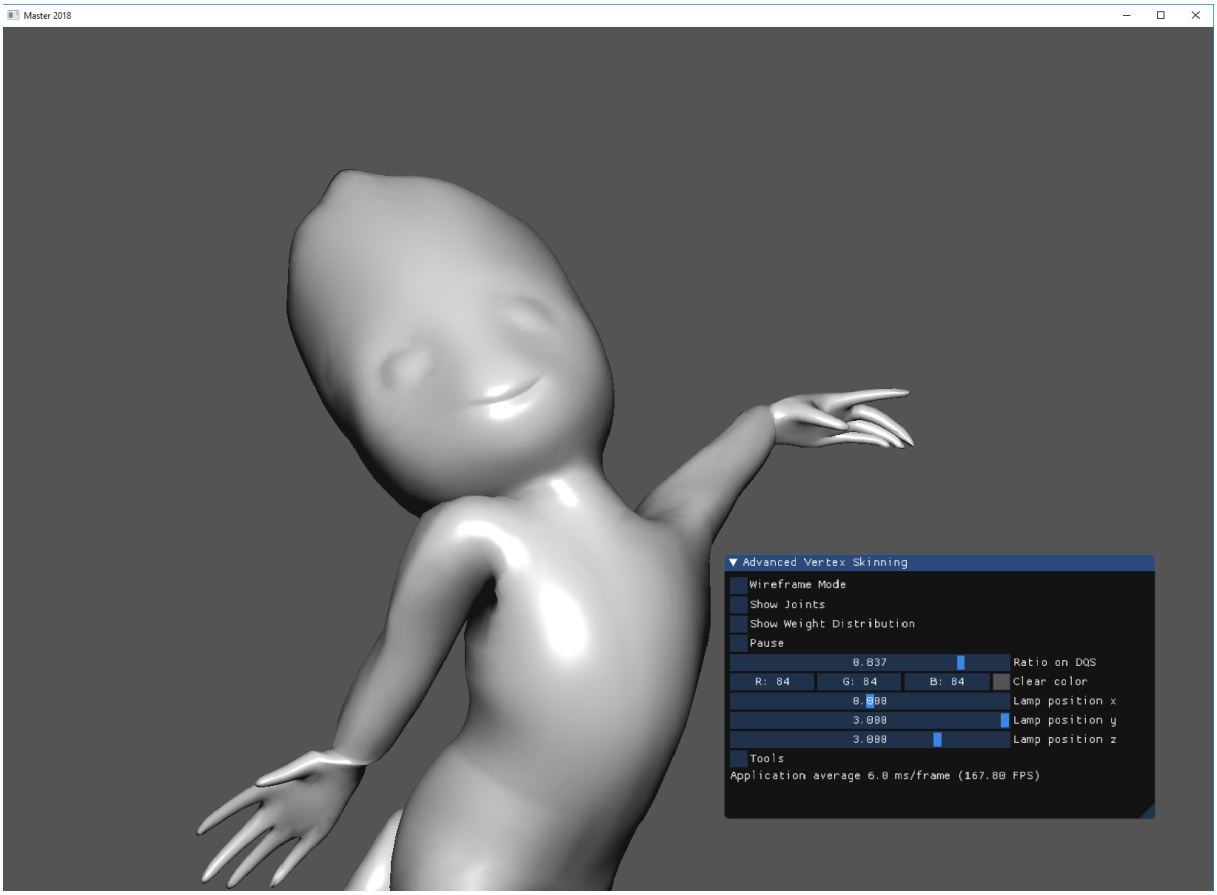


Figure 4.4. A screenshot of the developed OpenGL application window and the GUI controls.

4.7 Miscellaneous issues

As mentioned in section 4.2, the Blender software is employed for model validation. This step is performed because the models which we obtained lacked tail bones at the end of each bone chain. Such a tail bone is also known as "leaf bone" in rigging. Although lack of leaf bones does not affect animating the character, issues occur when visualising the bone segments on-screen as joints in the place of the end of bone chains that are missing. Therefore, we import all models to Blender and export to the .fbx files with the "add leaf bones" option, which adds end bones attached to every bone chain in the skeleton.

In the Model class discussed in section 4.4, the method loadMeshBones() stores the skinning weights of a bone. The weights are thresholded and set to 0.1 to eliminate weights whose contribution is negligible. This has the effect of reducing the amount of memory needed to store the skinning weights. When adding the weight, the associated vertex index is stored as well. ASSIMP stores the vertex index relevant to a single mesh, and in the application all meshes are kept in a single vector. Therefore, we calculate the absolute vertex index by adding the ASSIMP's vertex index to the number of vertices of previously processed meshes.

A difference in Kavan's original implementation [59] is that the components of a dual quaternion are in a different order, for example, the real part of the dual quaternion in our implementation is (w, x, y, z), while it corresponds to (x, y, z, w) in the original work. This is due to the construction of dual quaternions in the Model class using the GLM's dual quaternion extension, and the default order in the GLM library is (w, x, y, z) for constructing quaternions and dual quaternions. Therefore, in the vertex shader, this difference is adapted in the implementation of the DQtoMatrix() method.

Chapter V

Results and Comparative Analysis

This chapter outlines our results obtained by inspecting the deformation with several humanoid character's models and animations. We applied three models with different levels of complexity, described in table 5.1, to demonstrate our results and benchmark our proposed method. The skinning weights are specified by model creators and embedded in the model files. The Mutant and Starkie models contain textures. All models carry a sequence of keyframes animation, which are different from each other. In all comparisons, the same animation is performed on all skinning techniques.

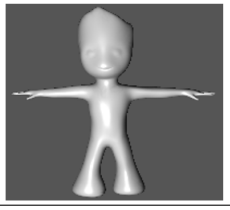
3D model	Mutant	Groo	Starkie
Rest pose			
Vertices	25,259	54,220	87,849
Triangles	11,271	27,146	90,048
Bones	43	53	65
Frames	136	506	716

Table 5.1. The models used in this research and their complexity

Section 5.1 provides the results of our experiment on DQS implementations, which were discussed in section 4.5.1. Section 5.2 provides objective verification of the improvements seen by using our method. Section 5.3 demonstrates the visual trade-off between the dual quaternion and linear blend skinning (DQS and LBS) influences when handling specific poses by varying the "deform factor". Section 5.4 provides further performance analysis derived from our implementation and discusses the implications.

5.1 DQS experiment result

This section presents the experiment result of the basic DQS implementation against antipodality handling DQS implementation. The theory of antipodality of quaternions and dual quaternions is discussed in section 2.1 and section 2.2, and the detail of implementation is discussed in section 4.2.2.

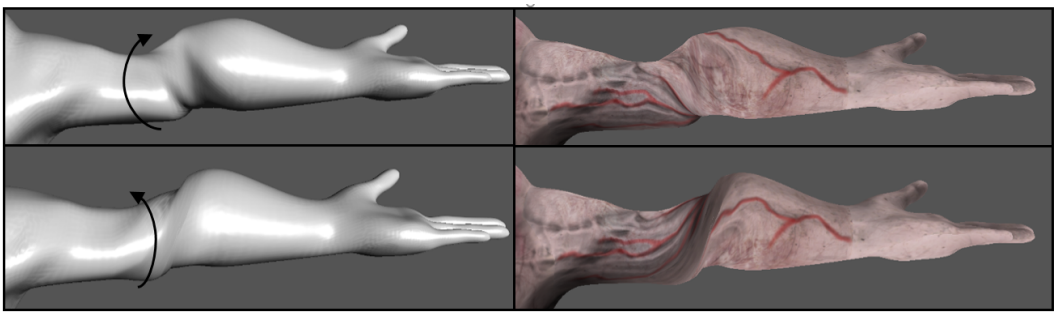


Figure 5.1. The basic DQS approach (top row) compared with the antipodality handling approach (bottom row). The antipodality handling resolves the distortion at the elbow and appears as a smooth dual quaternion blending, while the basic approach not only creates an undesired deformation but also causes discontinuities in the texture.

Per-vertex antipodality handling DQS generates a smooth dual quaternion blending at the joints, while the basic DQS implementation results in an artifact where the rotation at the joint is around the longer path. Figure 5.1 shows a comparison between different implementations from the deformation

obtained. With the same mesh model setup, the forearm is twisted through 180 degrees towards a counter-clockwise direction. The shoulder arm is not moved. In the case of basic DQS being performed, the rotation at the elbow produces a longer trajectory with a clockwise direction. The deformed mesh appears unnatural and discontinued. With textures attached, the artifact is more obvious. On the other hand, antipodality handling ensures the shortest rotation trajectory. The direction of the rotation appears unchanged (counter-clockwise). The resulting deformation is smooth and natural.

5.2 Correctness of our method

Our method aims to reduce the bulging artifact of the standard DQS. The bulges normally occur at bent joints and become more obvious with a significant bend, or at areas that are influenced by many joints. To reduce the bulges in these cases, we replaced half of the DQS result with the LBS result, by setting the "deform factor" to 0.5. In other words, we defined an equal amount of influence from the standard DQS and LBS techniques. We performed the proposed skinning algorithm on particular poses and compared them with the ground truth artifact generated from the standard DQS. The resulting deformation is improved and the bulges are sufficiently reduced.

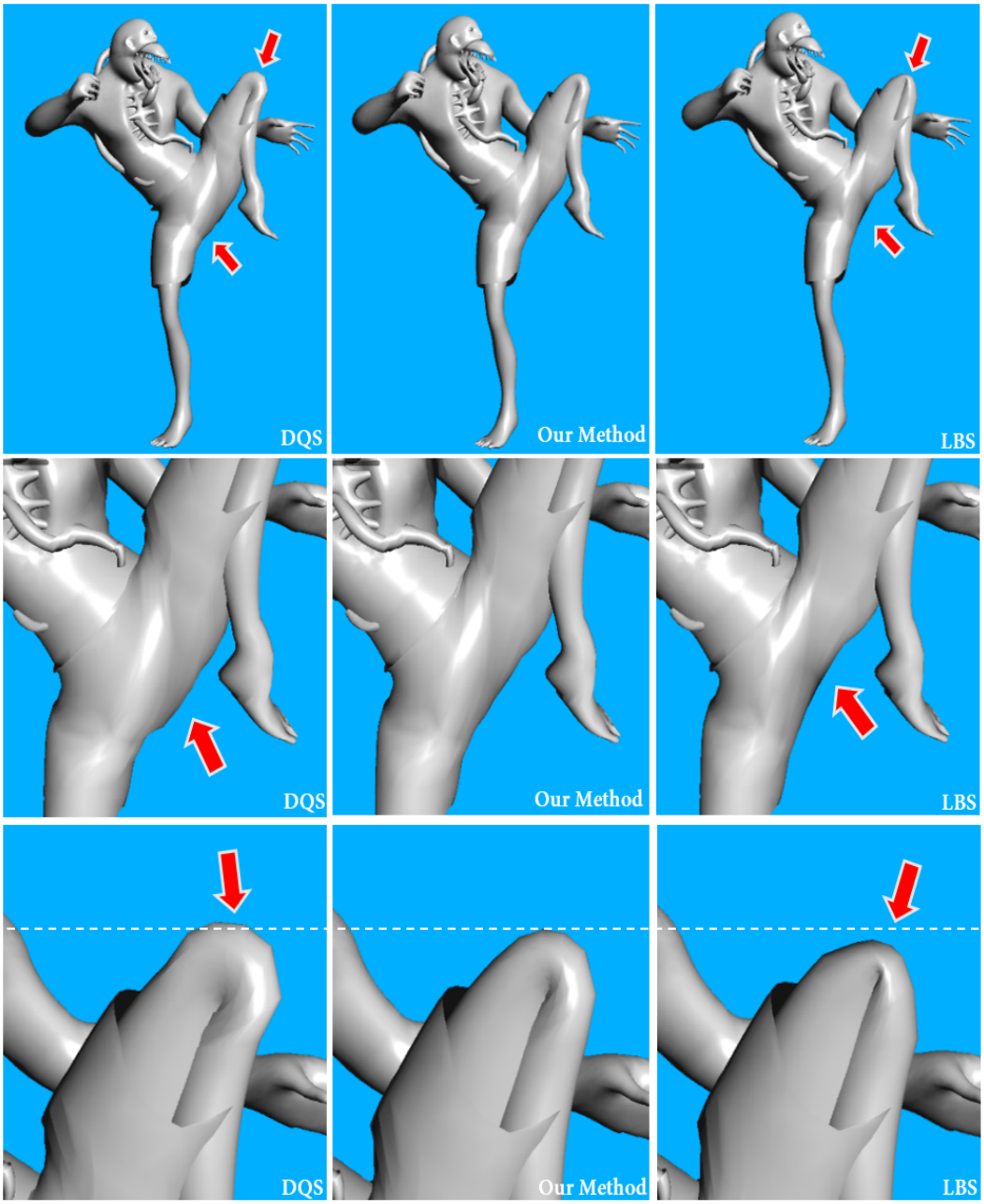


Figure 5.2. Extreme bends at the hip and the knee joints appear serious issues with both DQS (left) and LBS (right). Our method (middle) reduces the artifacts by applying an equal amount of DQS and LBS results.

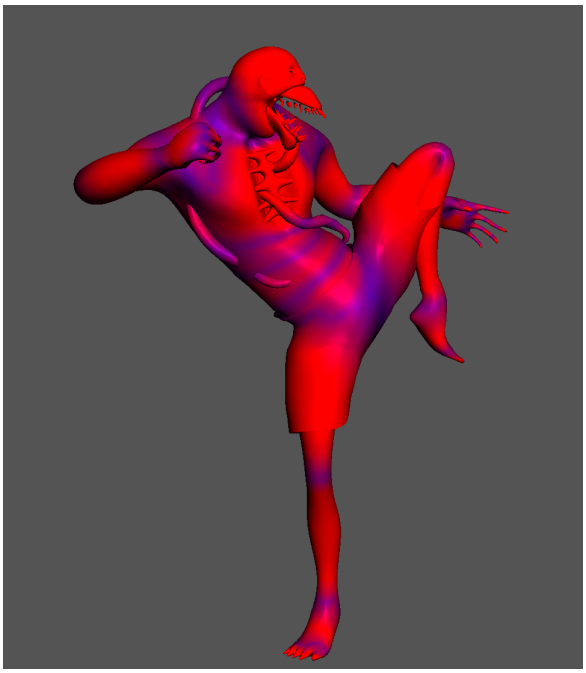


Figure 5.3. The weight distribution map of the Starkie model. Skinning weights equals to 1 correspond to red and 0 to blue.

Figure 5.2 shows our method improves the standard DQS at challenging places such as the hip and the knee. Extreme bends at the hip socket joints amplify the bulging artifact. The middle row represents the comparison between DQS, LBS, and the proposed skinning technique performed on the hip area of the Starkie model. The weight distribution map (see figure 5.3) manifests vertices at the hip area are affected by more than two bones. When DQS is performed, large rotations of the leg bones cause the interpolated hip mesh vertices position on the arc of a sphere, thus forming a curved-manifold, and this causes the deformed hip mesh to bulge outwardly. In contrast, LBS interpolates the vertices in the linear space, thus the interpolated vertices are located on a line segment, which results in an unsmooth and flattened deformed hip. Our method takes the equal amount of influences of DQS and LBS by setting the "deform factor" to 0.5. This indicates half of the DQS result is replaced with the LBS result. The interpolated vertices are

positioned at the middle of the DQS and LBS results and confer a more natural deformation at the hip.

Extreme bending at the knee joint appears as a serious issue with both DQS and LBS: DQS shows a large bulge, and LBS folds the knee too much, thus causing the "collapsing joint" artifact. The bottom row of figure 5.2 presents the comparison between DQS, LBS, and our method at the extremely bent knee of the Starkie model. The knee is influenced by two bones — the left lower leg and the left upper leg (see figure 5.3). The dotted line is drawn on the bottom row of figure 5.2 best illustrates the improvement of our method. Our method transforms the knee vertices to the middle of DQS and LBS results, while the DQS bulges push the knee above the line and the LBS collapses the knee severely, thus vertices are positioned below the line.

Figure 5.4. The weight distribution at the chest and the upper back muscle of the Mutant model. Skinning weights equals to 1 correspond to red and 0 to blue.

For vertices influenced by many joints (see figure 5.4), such as muscles at the chest and the upper back, the standard skinning algorithms produce

plausible deformations (see figure 5.5). DQS creates large bulges in the chest muscle, and the deformed chest appears rounder and unnatural. On the other hand, LBS flattens the edge around the chest, especially near the shoulder. At the upper back, DQS generates a visually distracting bulging artifact, while LBS appears as the "candy-wrapper" artifact that shrinks the volume of the upper back muscles.

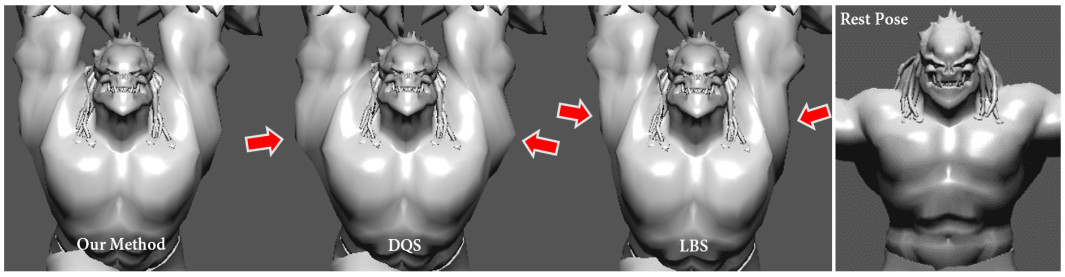


Figure 5.5. A comparison on the Mutant model with big muscles: with the "deform factor" set to 0.5, our method (left) reduces the "bulging joint" artifact of DQS (middle) on the chest and the "candy-wrapper" artifact of LBS (right) around the shoulder.

Our proposed method neutralises the DQS and LBS results by applying 0.5 to the "deform factor", thus reducing both DQS and LBS artifacts by half. Compared to the post-processing bulge-free method proposed by Kim et al. [1], our method generates the equivalent bulge-free deformation without projecting back the vertex positions after DQS transforms mesh vertices. And compared to Disney's skinning with pre-computed optimised CoRs technique [10], our method produces a comparable result with a more straightforward real-time approach. That is, instead of correcting the deformed mesh before or after the skinning process, our approach improves the artifacts while the vertex skinning is performing. The mesh vertices are transformed directly with a hybrid of DQS and LBS transformations. Even though our method is approximate, it minimises the artifacts inherent in DQS and LBS with fairly inexpensive cost while both previous corrective techniques [1] [10] re-

quire complex calculations. Upgrading an existing animation system from standard vertex skinning to the proposed skinning is simple and still permits an efficient GPU implementation.

However, because our method is a linear combination of DQS and LBS transformations, the LBS artifact is re-introduced. One solution is to determine the maximum amount of LBS influence that a specific joint rotation can have so that the deformed mesh appears with a minimum loss of volume that is not generally visually distracting. In the next section, we visualise the trade-off between DQS and LBS using our method and present a discussion of our approach and Maya's Blend Smooth Skinning tool based on the trade-off.

5.3 Trade-off between DQS and LBS influences

Our method allows the interpolation parameter, the "deform factor", to be modified in real-time, and this presents a visual trade-off between DQS and LBS at each frame. Such trade-off shows the decrease of the DQS artifact increasing the LBS artifact. When the joints bent with large rotations, an equal amount of DQS and LBS influence is suggested. When the bones are twisted with large rotations, the LBS influence is recommended to be close to zero. To demonstrate, the standard DQS is performed on bent and twisted elbows. Progressively reducing the DQS influences to a 0 result and the skinning turns to a pure LBS process.

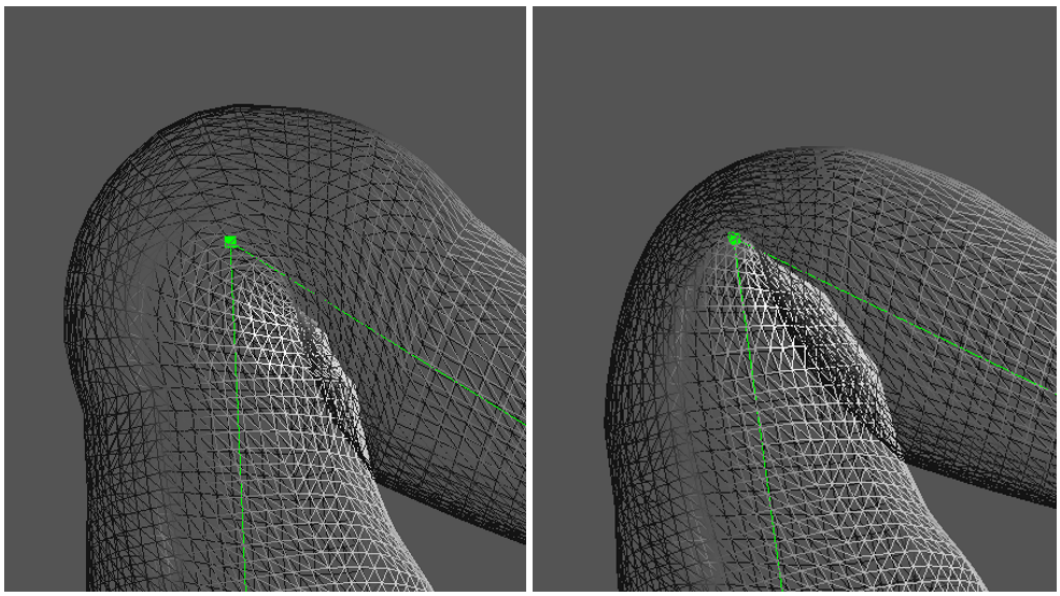


Figure 5.6. The bent elbow with 120 degrees rotation, resulting from DQS (left) and LBS (right). The green dot represents the elbow joint.

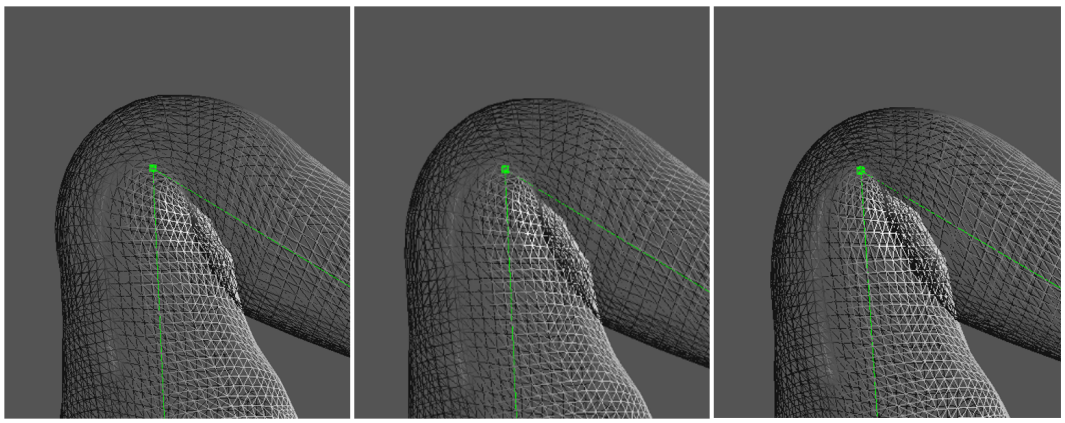


Figure 5.7. The deformed elbow with three corresponding values of the "deform factor": 0.75, 0.55 and 0.25 (left to right).

Figure 5.6 shows a rotation at the elbow of the Starkie model in wire-frame mode. When DQS is performed, the vertices on the bulged elbow mesh move further away from the joint (the elbow). In contrast, when LBS is performed, the vertices are moved closer to the joint. As shown in fig-

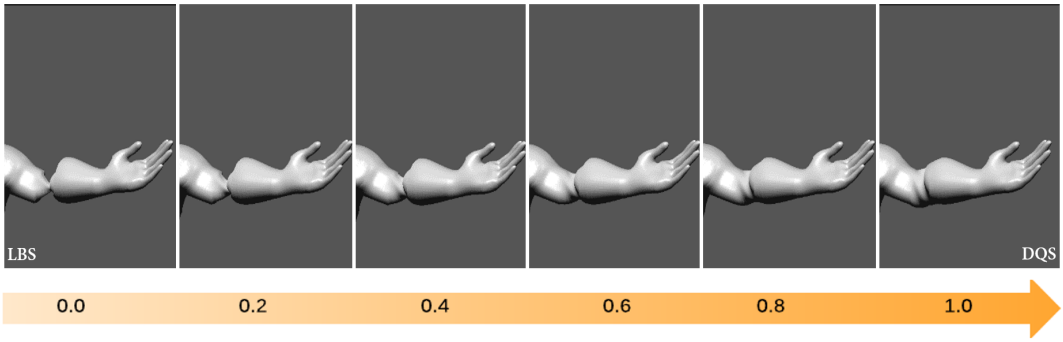


Figure 5.8. From left to right: The "candy-wrapper" artifact reduces gradually with the increase of the "deform factor". The values on the right arrow correspond to the influence of the standard DQS.

ure 5.7, vertices move towards the joint with the decreasing "deform factor", and when the "deform factor" is 0.55 the resulting deformation appears the smoothest and most natural. Therefore, for the bent elbow pose the ideal influence of DQS and LBS is approximately an equal amount of both.

Twisting an elbow is quite different from the bent elbow case. Twisting with LBS produces the "candy-wrapper" artifact, and DQS resolves the issue completely. When skinning with a combination of DQS and LBS, the volume loss of LBS is re-introduced to the deformed mesh, thus the "deform factor" is expected to be close to 1.0 and preserves the mesh volume as much as possible. Figure 5.8 shows the variations of the "candy-wrapper" artifact when the "deform factor" changes. In this example the forearm is rotated through 180 degrees while the shoulder arm is not moved. The "candy-wrapper" artifact appears at the elbow and becomes less visually observable when the "deform factor" becomes greater than 0.8.

In skinning applications, a particular pose is likely to have both twisting and bending in a single frame. Our method is constrained by the fact that all joints are assigned to a unified influence. When significant twisting and bending exists in one frame, this shortcoming can be reduced from ap-

plying a high DQS influence as the "candy-wrapper" artifact is more visually distracting than the "bulging joint" artifact.

Maya's Blend Smooth Skinning tool does not have such constraints on joint influences. It allows artists to define different impacts on each joint for every pose. This manual procedure results in more accurate skin deformation around joints compared to our method and provide artists with more control in blending the skinning techniques. Nevertheless, the process is highly complex and iterative. Our proposed method enables the contribution of DQS and LBS to be modified according to the instantaneous visual feedback the artist receives while performing the vertex skinning. The balance between DQS and LBS influences is where both artifacts are minimised concerning the compromise between the two standard methods, and it is ordinarily subjective from the 3D artists.

Compared to Maya's tool, our method allows a non-expert user to perform basic vertex skinning. Also, our approach saves the time and effort that users spend on multiple iterations of painting the blend weight map. The unified influence constraints implicate our method is not suitable for applications that have a high requirement of skin deformation accuracy. Instead, it is ideal for mobile games and VR applications. One concern of the proposed method is the robustness of linearly combining two standard skinning algorithms in real-time. In the next section, we discuss the performance analysis of the proposed approach.

5.4 Performance and memory requirements of our method

Table 5.2 summarises the memory requirement of the proposed method on the models and animations we used.

Vertex skinning algorithm	Per-joint scalar	VBO	Uniform variable
LBS	12	64 bytes	48
DQS	8	64 bytes	32
Ours	20	64 bytes	80

Table 5.2. The memory requirement comparison

3D models	LBS	DQS	Ours
Mutant	6.23 ms/f	6.07 ms/f	6.14 ms/f
Groo	8.95 ms/f	8.89 ms/f	8.44 ms/f
Starkie	9.45 ms/f	9.21 ms/f	9.12 ms/f

Table 5.3. Performance result of three vertex skinning algorithms

Our method to linearly combine DQS with LBS requires 20 scalars per joint, compared with the 12 expected by LBS and the eight required by DQS. The VBO for the standard LBS and DQS, and our method is composed of vertex positions (three floats), vertex normals (three floats), texture coordinates (two floats), bone weights (four floats), and bone indices (four ints), summing up to 64 bytes. The uniform variable per bone is the 3x4 float matrix (48 bytes) for LBS and the 2x4 float matrix (32 bytes) for DQS. Our method requires one more float uniform which is the "deform factor", which sums up to 80 bytes. The shader instruction counts of the proposed algorithm is directly the sum of DQS and LBS shader instruction counts.

Each skinning algorithm is performed on the Groo, Starkie, and Mutant models with the same setup. Each model comprises a different set of keyframes animation. We estimate the run-time performance from a collection of 3000 frames recorded per model. The averaged time cost measures run-time performance in milliseconds of each frame. As shown in table 5.3,

for all three models, the performance of our proposed algorithm is mostly similar to DQS and LBS due to the effectiveness of GPU implementation. Precisely, the proposed algorithm is down by less than 1 ms/frame in all cases compared to standard skinning algorithms.

Figures 5.9, 5.10, and 5.11 indicate the robustness of the proposed method. The robustness is measured from 3000 frames for every three models with animation sequences, at five DQS contribution levels. The cost of time per frame is recorded when the "deform factor" is set to 0.0, 0.25, 0.55, 0.75, and 1.0, which corresponds to the amount of the DQS influences on the proposed vertex skinning algorithm. The result shows the time cost per frame is relatively similar at respective DQS contribution level. In the case of the Mutant model and the Starkie model, the lines correspond to the DQS contributions mostly overlap and remain stable. Although there are unpredictable rise and fall in the Groo model test, the overall trend of individual DQS contribution exhibits the performance is comparable. In general, the proposed method is proved to be reasonably robust while changing the influence of DQS in real-time.

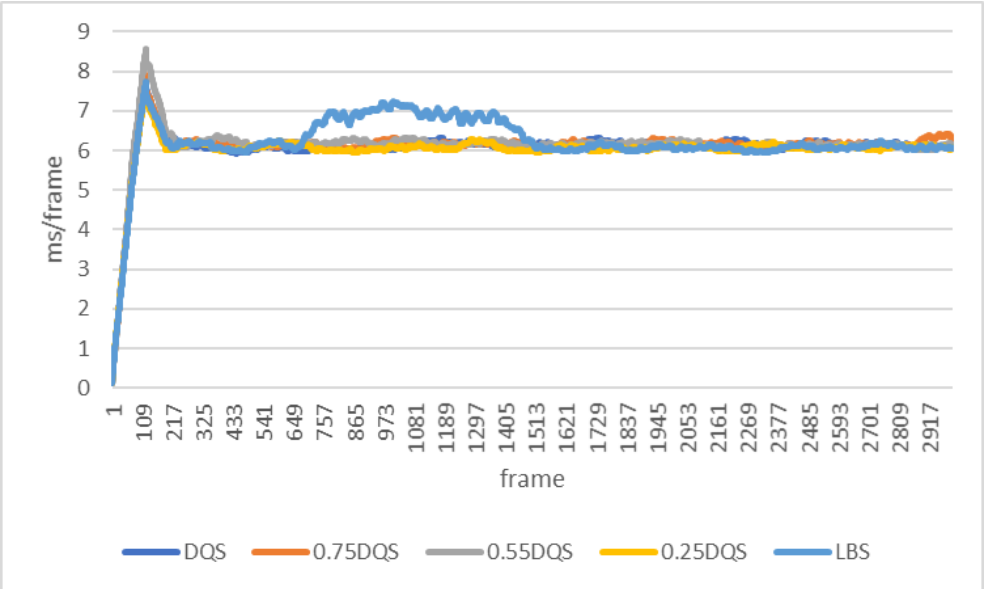


Figure 5.9. Robustness test on Mutant model

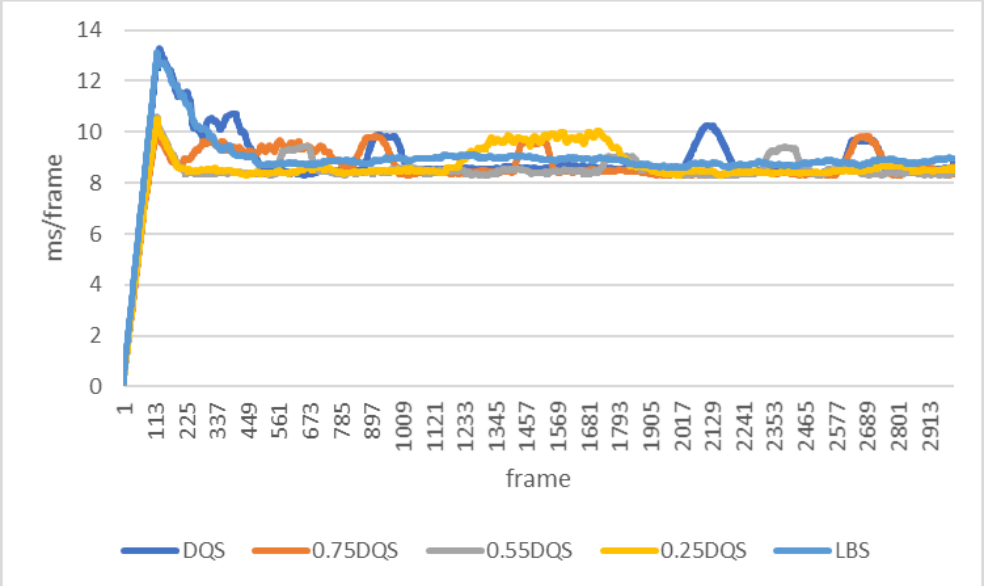


Figure 5.10. Robustness test on Groo model

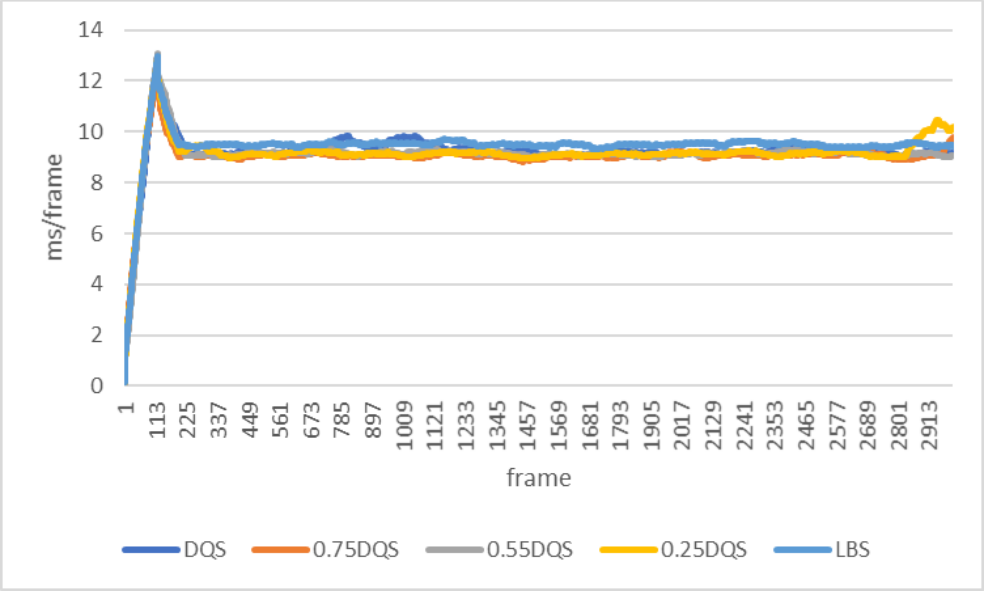


Figure 5.11. Robustness test on Starkie model

Chapter VI

Conclusions

6.1 Conclusion

This thesis conducts a detailed study of traditional vertex skinning techniques, with the main focus on vertex skinning with dual quaternions. The traditional skinning technique (DQS) produces noticeable bulges at joints with large rotations, while our method effectively removes such artifacts. Furthermore, the usefulness of a linear combination of DQS and LBS has not been researched before, and this thesis fills that gap by carrying out a systematic analysis.

A complete 3D character animation system is implemented in this thesis, and the proposed skinning algorithm is executed on the GPU with the same setup of traditional skinning techniques. The proposed algorithm is direct and closed-form, and reduce the artifacts without pre-processing of model information or post-processing of animated poses, which is the main advantage compared to other DQS corrective methods and Maya's Blend Smooth Skinning. The proposed skinning algorithm can be widely adopted in low-end real-time animation applications such as VR and mobile games because of its simplicity and high robustness.

6.2 Future work

This study only focused on rigid transformations, and non-rigid transformations such as scale and shear are not included in the scope of this the-

sis. Kavan's paper [9] presented a multi-phase process of handling non-rigid transformation, and the non-rigid factors are interpolated via LBS separately. Our work could be extended to support non-rigid transformations as LBS is performed in the vertex shader directly with the existing setup.

One of the required inputs of the proposed method is a set of predefined skinning weights specified by the artist. There are various skinning weight assignment schemes available, for example "bone glow" [70], or skinning weights that are designed mainly for DQS. Future works could experiment on different skinning weight assignment algorithms with the proposed skinning algorithm.

In this thesis, the value of the "deform factor" is manually adjusted, and the skinning technique contributions are applied to all joints uniformly. To provide more controls for the 3D artists, the value of the "deform factor" should ideally be set automatically with a non-uniformed approach. Kavan's paper [25] proposed an efficient method that automatically determines challenging areas to add extra blend bones, and future works could use this method for automatically setting the "deform factor" at different joints. This could be an appealing direction for future research to develop an optimised linear combination of DQS and LBS.

The trade-off between DQS and LBS is presented qualitatively in this thesis, and this would need to be verified with a perceptual study provided with quantitative analysis. One possible future direction is analysing the difference between the distance of joint area vertices to the joint at rest pose and animated pose, and comparing the displacement of the animated distance to the rest pose distance while adjusting the standard skinning influences. This could provide a quantitative analysis of the trade-off between DQS and LBS.

As an extension to the above future direction, the distance from ver-

tices to the nearest joint can be used to develop further a corrective DQS technique, which is similar to the approach in Kim's paper [1]. If a vertex is detected moving away from the joint, then only push it back to its rest pose position with regard to the nearest joint.

Similar to the standard DQS and LBS, the proposed skinning technique in this research could also be combined with example-based or physics-based skinning techniques to produce more complex skin deformations. So far, advanced skinning techniques have combined with LBS and DQS regularly, so that future work can build advanced techniques on the proposed method for secondary effect generation.

6.2.1 Conference publication

We aim to publish our algorithm and experimental results in an upcoming conference on Augmented Reality, Virtual Reality and Computer Graphics to be held in Santa Maria al Bagno in Italy. [71].

Appendix A

DQS experiment result: lighting artifact

As discussed in section 4.5.1, if the second possible approach of our implementation adopts Kavan's optimised DQS implementation, it will lead to a linear interpolation of vertex normal vector. However, the interpolation introduces an unexpected lighthing artifact and becomes more observable with larger rotations. Figure A.1 shows the artifact when twisting the elbow via the optimised DQS approach and linearly blend the normal vectors with the possible approach.

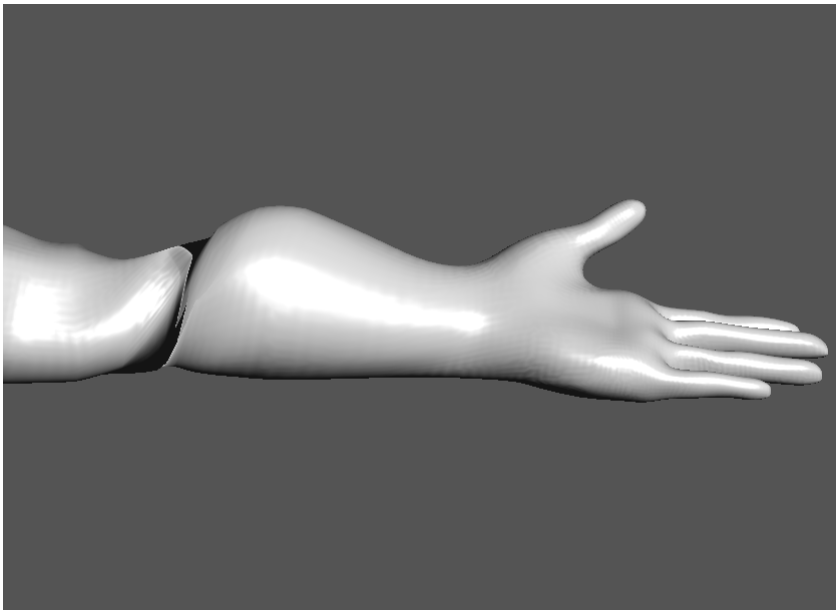


Figure A.1. Linearly interpolating the normal vectors causing a serious lighting artifact with large rotations.

References

[1] Y. Kim and J. Han, "Bulging-free dual quaternion skinning," *Computer Animation and Virtual Worlds*, vol. 25, no. 3-4, pp. 321–329, 2014.

[2] J. P. Lewis, M. Cordner, and N. Fong, "Pose space deformation: A unified approach to shape interpolation and skeleton-driven deformation," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, (New York, NY, USA), pp. 165–172, ACM Press/Addison-Wesley Publishing Co., 2000.

[3] L. Liu, K. Yin, B. Wang, and B. Guo, "Simulation and control of skeleton-driven soft body characters," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 6, p. 215, 2013.

[4] A. McAdams, Y. Zhu, A. Selle, M. Empey, R. Tamstorf, J. Teran, and E. Sifakis, "Efficient elasticity for character skinning with contact and collisions," *ACM Trans. Graph.*, vol. 30, pp. 37:1–37:12, July 2011.

[5] "Maya's Blend Smooth Skinning." `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/Maya/files/GUID-B292915D-EA18-43DA-A817-58DDC3DF879D-htm.html`. Accessed: 2019-02-15.

[6] B. Kenwright, "A beginners guide to dual-quaternions: what they are, how they work, and how to use them for 3d character hierarchies," 2012.

[7] A. J. Hanson, "Visualizing quaternions," in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), ACM, 2005.

[8] K. Shoemake, "Animating rotation with quaternion curves," *SIGGRAPH Comput. Graph.*, vol. 19, pp. 245–254, July 1985.

[9] L. Kavan, S. Collins, J. Zara, and C. O'Sullivan, "Geometric skinning with approximate dual quaternion blending," *ACM Trans. Graph.*, vol. 27, no. 4, p. 105, 2008.

[10] B. H. Le and J. K. Hodgins, "Real-time skeletal skinning with optimized centers of rotation," *ACM Trans. Graph.*, vol. 35, pp. 37:1–37:10, July 2016.

[11] W. K. Clifford, *Mathematical papers*. Macmillan and Company, 1882.

[12] R. Mukundan, *Advanced methods in computer graphics: with examples in OpenGL*. Springer Science & Business Media, 2012.

[13] "Dual quaternion wikipedia page." `https://en.wikipedia.org/wiki/Dual_quaternion`. Accessed: 2019-02-23.

[14] L. Kavan, S. Collins, C. OSullivan, and J. Zara, "Dual quaternions for rigid transformation blending," *Technical report, Trinity College Dublin*, 2006.

[15] H.-L. Pham, V. Perdereau, B. V. Adorno, and P. Fraisse, "Position and orientation control of robot manipulators using dual quaternion feedback," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pp. 658–663, IEEE, 2010.

[16] M. Schilling, "Universally manipulable body modelsdual quaternion representations in layered and dynamic mmcs," *Autonomous Robots*, vol. 30, no. 4, p. 399, 2011.

[17] A. Jacobson, Z. Deng, L. Kavan, and J. Lewis, "Skinning: Real-time shape deformation," in *ACM SIGGRAPH*, vol. 22, 2014.

[18] E. Catmull, "A system for computer generated movies," in *Proceedings of the ACM annual conference-Volume 1*, pp. 422–431, ACM, 1972.

[19] N. Magnenat-thalmann, R. Laperrire, D. Thalmann, and U. D. Montral, "Joint-dependent local deformations for hand animation and object grasping," in *In Proceedings on Graphics interface 88*, pp. 26–33, 1988.

[20] M. Alexa, "Linear combination of transformations," *ACM Trans. Graph.*, vol. 21, pp. 380–387, July 2002.

[21] N. Magnenat-Thalmann, F. Cordier, H. Seo, and G. Papagianakis, "Modeling of bodies and clothes for virtual environments," in *2004 International Conference on Cyberworlds*, pp. 201–208, Nov 2004.

[22] L. Kavan and J. Zara, "Real-time skin deformation with bones blending," in *WSCG Short Papers Proceedings*, 2003.

[23] C. Bloom, J. Blow, and C. Muratori, "Errors and omissions in marc alexas linear combination of transformations," 2004.

[24] X. Yang and J. J. Zhang, "Stretch it-realistic smooth skinning," in *International Conference on Computer Graphics, Imaging and Visualisation (CGIV'06)*, pp. 323–328, IEEE, 2006.

[25] L. Kavan, S. Collins, and C. O'Sullivan, "Automatic linearization of nonlinear skinning," in *2009 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 49–56, ACM Press, February/March 2009.

[26] J. Hejl, "Hardware skinning with quaternions," in *Game Programming Gems 4* (A. Kirmse, ed.), pp. 487–495, Charles River Media, 2004.

[27] L. Kavan and J. Zara, "Spherical blend skinning: A real-time deformation of articulated models," in *2005 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 9–16, ACM Press, April 2005.

[28] L. Kavan, S. Collins, J. Zara, and C. O'Sullivan, "Skinning with dual quaternions," in *2007 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 39–46, ACM Press, April/May 2007.

[29] G. S. Lee, A. Lin, M. Schiller, S. Peters, M. McLaughlin, F. Hanner, and W. D. A. Studios, "Enhanced dual quaternion skinning for production use.," in *SIGGRAPH Talks*, pp. 9–1, 2013.

[30] "Bulge-free DQS issue." `http://rodolphe-vaillant.fr/?e=72`. Accessed: 2019-02-15.

[31] L. Kavan and O. Sorkine, "Elasticity-inspired deformers for character articulation," *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH ASIA)*, vol. 31, no. 6, pp. 196:1–196:8, 2012.

[32] R. Vaillant, L. Barthe, G. Guennebaud, M.-P. Cani, D. Rohmer, B. Wyvill, O. Gourmel, and M. Paulin, "Implicit skinning: real-time skin deformation with contact modeling," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, p. 125, 2013.

[33] R. Vaillant, G. Guennebaud, L. Barthe, B. Wyvill, and M.-P. Cani, "Robust iso-surface tracking for interactive character skinning," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 6, p. 189, 2014.

[34] "Blender software manual, skinning." `https://docs.blender.org/manual/en/latest/rigging/armatures/skinning/introduction.html`. Accessed: 2019-02-15.

[35] "Autodesk 3ds Max, skinning a character." `https://knowledge.autodesk.com/support/3ds-max/getting-started/caas/CloudHelp/cloudhelp/2018/ENU/3DSMax-Tutorial/files/GUID-293EE899-638B-459F-AFFC-F6302DEE3305-htm.html`. Accessed: 2019-02-15.

[36] "Maya Smooth Skinning, skinning your character." `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-CharacterAnimation/files/GUID-EFE68C08-9ADA-4355-8203-5D1D109DCC82-htm.html`. Accessed: 2019-02-15.

[37] P.-P. J. Sloan, C. F. Rose III, and M. F. Cohen, "Shape by example," in *Proceedings of the 2001 symposium on Interactive 3D graphics*, pp. 135–143, ACM, 2001.

[38] T. Kurihara and N. Miyata, "Modeling deformable human hands from medical images," in *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '04, (Goslar Germany, Germany), pp. 355–363, Eurographics Association, 2004.

[39] P. G. Kry, D. L. James, and D. K. Pai, "Eigenskin: Real time large deformation character skinning in hardware," in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '02, (New York, NY, USA), pp. 153–159, ACM, 2002.

[40] T. Rhee, J. P. Lewis, and U. Neumann, "Real-time weighted pose-space deformation on the gpu," in *Computer Graphics Forum*, vol. 25, pp. 439–448, Wiley Online Library, 2006.

[41] R. Y. Wang, K. Pulli, and J. Popović, "Real-time enveloping with rotational regression," *ACM Trans. Graph.*, vol. 26, July 2007.

[42] M. Loper, N. Mahmood, J. Romero, G. Pons-Moll, and M. J. Black, "Smpl: A skinned multi-person linear model," *ACM transactions on graphics (TOG)*, vol. 34, no. 6, p. 248, 2015.

[43] X. C. Wang and C. Phillips, "Multi-weight enveloping: Least-squares approximation techniques for skin animation," in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '02, (New York, NY, USA), pp. 129–138, ACM, 2002.

[44] H. Pyun, Y. Kim, W. Chae, H. W. Kang, and S. Y. Shin, "An example-based approach for facial expression cloning," in *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, (New York, NY, USA), ACM, 2006.

[45] S. I. Park and J. K. Hodgins, "Data-driven modeling of skin and muscle deformation," in *ACM Transactions on Graphics (TOG)*, vol. 27, p. 96, ACM, 2008.

[46] X. Shi, K. Zhou, Y. Tong, M. Desbrun, H. Bao, and B. Guo, "Example-

based dynamic skinning in real time," *ACM Trans. Graph.*, vol. 27, pp. 29:1–29:8, Aug. 2008.

[47] G. S. Lee and F. Hanner, "Practical experiences with pose space deformation," in *ACM SIGGRAPH ASIA 2009 Sketches*, SIGGRAPH ASIA '09, (New York, NY, USA), pp. 43:1–43:1, ACM, 2009.

[48] D. Komorowski, V. Melapudi, D. Mortillaro, and G. S. Lee, "A hybrid approach to facial rigging," in *ACM SIGGRAPH ASIA 2010 Sketches*, SA '10, (New York, NY, USA), pp. 42:1–42:2, ACM, 2010.

[49] C. Schumacher, B. Thomaszewski, S. Coros, S. Martin, R. Sumner, and M. Gross, "Efficient simulation of example-based materials," in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '12, (Goslar Germany, Germany), pp. 1–8, Eurographics Association, 2012.

[50] J. S. Zurdo, J. P. Brito, and M. A. Otaduy, "Animating wrinkles by example on non-skinned cloth," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 1, pp. 149–158, 2013.

[51] B. H. Le and Z. Deng, "Robust and accurate skeletal rigging from mesh sequences," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, p. 84, 2014.

[52] "Maya muscle plugin description." `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-CharacterAnimation/files/GUID-90B5E302-8DAA-4780-BCD6-FB9C60FF9E05-htm.html`. Accessed: 2019-02-17.

[53] "Weta digital's tissue technology." https://www.wetafx.co.nz/research-and-tech/technology/tissue/. Accessed: 2019-02-17.

[54] O. Rémillard and P. G. Kry, "Embedded thin shells for wrinkle simulation," *ACM Trans. Graph.*, vol. 32, pp. 50:1–50:8, July 2013.

[55] D. Li, S. Sueda, D. R. Neog, and D. K. Pai, "Thin skin elastodynamics," *ACM Trans. Graph.*, vol. 32, pp. 49:1–49:10, July 2013.

[56] M. Kim, G. Pons-Moll, S. Pujades, S. Bang, J. Kim, M. J. Black, and S.-H. Lee, "Data-driven physics for human soft tissue animation," *ACM Trans. Graph.*, vol. 36, pp. 54:1–54:12, July 2017.

[57] R. Luo, W. Xu, H. Wang, K. Zhou, and Y. Yang, "Physics-based quadratic deformation using elastic weighting," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 12, pp. 3188–3199, 2018.

[58] K. Zhou, M. Chai, and C. Zheng, "Real-time animation method for hair-object collisions," Sept. 20 2018. US Patent App. 15/533,297.

[59] "Kavan's implementation of dual quaternion skinning vertex shaders." https://www.cs.utah.edu/~ladislav/dq/dqs.cg. Accessed: 2019-02-15.

[60] "The concept of linear combination." https://en.wikipedia.org/wiki/Linear_combination. Accessed: 2019-02-17.

[61] "GLFW homepage, an opengl library." https://www.glfw.org/. Accessed: 2019-02-15.

[62] "GLEW homepage, the opengl extension wrangler library." https://www.glfw.org/. Accessed: 2019-02-15.

[63] "Mixamo 3d charaters and animations." https://www.mixamo.com/#/. Accessed: 2019-02-15.

[64] "Sketchfab homepage." https://sketchfab.com/feed. Accessed: 2019-02-15.

[65] "Blender homepage." https://www.blender.org/. Accessed: 2019-02-15.

[66] "ASSIMP open asset import library." http://www.assimp.org/. Accessed: 2019-02-15.

[67] "GLM homepage." https://glm.g-truc.net/0.9.9/index.html. Accessed: 2019-02-15.

[68] "Dear ImGui: Bloat-free Immediate Mode Graphical User interface for C++ with minimal dependencies." https://github.com/ocornut/imgui. Accessed: 2019-02-15.

[69] "Single-file public domain (or MIT licensed) libraries for C/C++." https://github.com/nothings/stb. Accessed: 2019-02-15.

[70] R. Wareham and J. Lasenby, "Bone glow: An improved method for the assignment of weights for mesh deformation," in *International Conference on Articulated Motion and Deformable Objects*, pp. 63–71, Springer, 2008.

[71] "6th international conference on augmented reality, virtual reality and computer graphics." http://www.salentoavr.it/. Accessed: 2019-02-21.

[72] B. Rim and L. Schiaratura, "Gesture and speech in fundamentals of nonverbal behavior," 01 1991.

[73] E. Coronado, J. Villalobos, B. Bruno, and F. Mastrogiovanni, "Gesture-based Robot Control: Design Challenges and Evaluation with Humans," 05 2017.

[74] A. Mohr and M. Gleicher, "Building efficient, accurate character skins from examples," in *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, (New York, NY, USA), pp. 562–568, ACM, 2003.